hochschule mannheim

Masterarbeit

Konzeption eines Blockchain-basierten Triple-A-Systems und Implementierung als EAP-Erweiterung zur Authentifizierung über IEEE 802.1X

Fakultät für Informationstechnik
Institut für Softwaretechnik und Datenkommunikation

David Amann
Masterstudiengang Informationstechnik

Betreuender Professor: Prof. Dr. Jens-Matthias Bohli
Zweitkorrektor: M.Sc. Christian Armbrust

Zeitraum: 1. April 2017 – 30. September 2017

Erklärung

Hiermit erkläre ich, die vorliegende Masterarbeit selbst angefertigt und alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben zu haben. Ich habe alles kenntlich gemacht, was ich aus Arbeiten anderer unverändert oder mit Änderungen übernommen habe.

Mannheim, 27.9.2017
David Amann

Abstrakt

In dieser Masterarbeit ist die Konzeption eines Blockchain-basierten Triple-A-Systems und der Implementierung einer Erweiterung des Extensible Authentication Protocol (EAP; deutsch: Erweiterbares Authentifizierungsprotokoll) [RFC3748][8] in Form einer EAP-Methode beschrieben. Die EAP-Methode wird hierbei als Authentifizierungsprotokoll für den WLAN-Zugriff nach IEEE 802.1X [RFC3580][12] genutzt.

Die Blockchain Ethereum[37] bildet dabei die Plattform für das Backend des Triple-A-Systems (AAA-Systems; Authentifizierungs-, Autorisierungs- und Abrechnungs-Systems), in Form eines Smart-Contracts.

Abbildungsverzeichnis

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziele und Anforderungen

Das primäre Ziel dieser Arbeit ist die Konzeption eines Blockchain-basierten Triple-A-Systems (AAA-Systems) und die anschliessende Implementierung einer EAP-Methode (EAP-Erweiterung) mit der es ermöglicht wird, eine Authentifizierung für einen Netzwerkzugriff mit dem Blockchain-basierten Triple-A-System durchzuführen. Die Authentifizierung für den Netzwerkzugriff, wird dabei über den Standard *IEEE 802.1X* durchgeführt.

Als sekundäres Ziel ist die Entwicklung eines Ansatzes zur automatischen Verrechnung (Payment) im Triple-A-System angedacht. Dabei ist die Idee, die Kosten der Netzwerknutzung automatisch über die Blockchain zu abzurechnen.

Damit soll die Frage geklärt werden, ob und wie genau die Blockchain-Technologie für den Einsatz in diesem Gebiet der Informationstechnik nutzbar ist.

Eine Anforderung an alle in dieser Arbeit verwendeten Kryptosysteme ist, ein Sicherheitsniveau von mindesten 128 Bit einzuhalten. Dies soll eine ausreichende Sicherheit gewährleisten während der Einsatzdauer des Authentifizierungsprotokolls.

1.2 Die Blockchain

Die Blockchain (auch Blockchain-Technologie) ist die Grundlage moderner auf kryptologischen Verfahren basierender Währungen. Entstanden ist die Blockchain-Technologie mit der ersten populären Kryptowährung Bitcoin bzw. in dessen zugrunde liegender Publikation von Satoshi Nakamoto *Bitcoin: A Peer-to-Peer Electronic Cash System*[25].

Eine Blockchain kann als kryptografisch gesicherte, dezentralisierte Datenbank betrachtet werden. Diese speichert, bei Kryptowährungen, Transaktion um, anhand dieser einen aktuellen Status des gesamten Kryptowährungs-Netzwerkes abzubilden und somit das mehrfache Ausgeben von kryptografischem Kapital zu verhindern.

Die Blockchain besteht dabei im Wesentlichen aus einzelnen miteinander verketteten (Daten-) Blöcken, daher auch der englische Name *block chain*. Jeder Block, mit Ausnahme des ersten Blocks (sogenannter *Genesis Block*), verweist dazu auf den vorherigen.

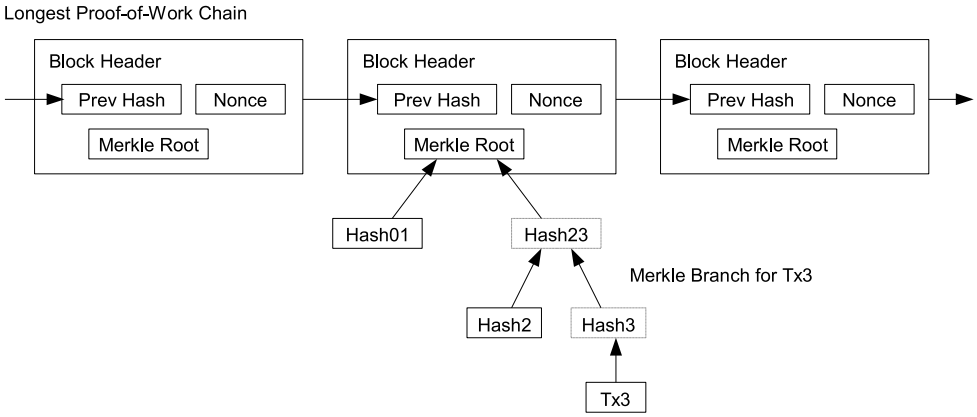


Abbildung 1.1: *Verkettung von Blöcken am (vereinfachten) Beispiel von Bitcoin [25]*

Jeder Teilnehmer des Blockchain-Netzwerks speichert eine Kopie der Blockchain. Der Datenaustausch, zur Synchronisation der Daten der Blockchain, erfolgt über ein Peer-to-Peer (P2P) Netzwerk, bei dem jeder Teilnehmer der Blockchain ein Mitglied des darunterliegenden Peer-to-Peer Netzwerks ist.

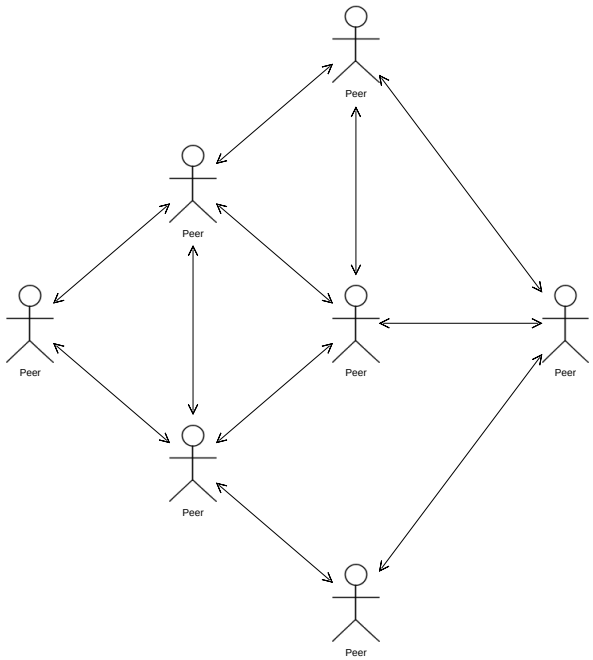


Abbildung 1.2: *Beispiel eines Peer-to-Peer Netzwerks*

Welchen Nutzen erfüllt nun die Blockchain?

Zum Erstellen digitaler Transaktionen bedarf es (noch) keiner Blockchain! Transaktionen können auch ohne die Blockchain-Technologie, mithilfe eines asymmetrischen Kryptosy-

stems (Public-Key-Kryptosystems), mit einer digitalen Signatur versehen werden und somit von allen Teilnehmern eines Netzwerks validiert werden.

Das Problem ist, dass immer eine Instanz benötigt wird, die verhindert, dass das kryptografische Geld mehr als einmal ausgegeben wird. Diese Instanz kann eine vertrauenswürdige Institution sein, die eine Kopie aller Transaktionen speichert und Teilnehmern des Kryptowährungssystems erlaubt die Gültigkeit von neuen Transaktionen zu überprüfen, um sicherzustellen, dass das kryptografische Geld das mit der Transaktion auf einen anderen Teilnehmer übergehen soll, noch existiert und nicht schon zu einem anderen Teilnehmer transferiert wurde. Somit wird die Integrität des gesamten Kryptowährungssystems gewährleistet.

An diesem Punkt kommt die Blockchain-Technologie zum Einsatz. Die Blockchain ermöglicht es, die Aufgabe der vertrauenswürdigen Institution auf jeden Teilnehmer des Kryptowährungssystems zu verteilen. Somit gibt es keine einzelne Institution, der man bedingungslos vertrauen muss.

1.2.1 Funktionsweise der Blockchain

Im übertragenen Sinn kann eine Blockchain als ein verteiltes Kontobuch gesehen werden. Jeder, der an diesem Kontobuch beteiligt ist, führt eine Kopie des Kontobuchs. Und jeder Beteiligte ist Teilnehmer eines vermachten Netzwerks (Peer-to-Peer Netzwerk). Dieses Netzwerk dient dazu, die jeweiligen Kopien des Kontobuchs untereinander aktuell zu halten bzw. auf dem gleichen Stand zu halten. Jeder Teilnehmer ist immer nur mit einer Untermenge aller Teilnehmer des Blockchain-Netzwerkes verbunden bzw. kann sich mit diesen Austauschen.

Das Kontobuch wird immer seitenweise ergänzt. Jede Seite in dem Kontobuch entspricht einem Block in der Blockchain. Somit können die Seitennummer und die Nummer eines Blocks gleichgesetzt werden.

Jede Seite enthält Überweisungen, sogenannte Transaktionen. Die Transaktionen selbst enthalten (vereinfacht) eine Liste mit Auftraggeber- (Schuldner-) Konten, mit den jeweiligen von dessen Konten abzuziehenden Beträgen und einer bestätigenden Unterschrift, sowie eine List mit Empfänger- (Gläubiger-) Konten mit den jeweiligen gutzuschreibenden Beträgen. Die Summe der zu transferierenden Beträge muss auf der Auftraggeberseite mindestens so groß sein wie auf der Empfängerseite. Alles, was darüber hinaus geht, kann als Transaktionsgebühr betrachtet werden. Alle Transaktionsgebühren einer Seite (Blocks) sind in der Summe eine Vergütung, die der Ersteller der Kontobuch-Seite erhält.

Möchte ein Teilnehmer eine Überweisung durchführen, erstellt er eine Transaktion und versendet diese an alle mit ihm verbundenen Teilnehmer des Blockchain-Netzwerkes.

Jeder Teilnehmer, der eine Transaktion zugesandt bekommt, überprüft diese. Sollte sie nicht den Regeln der Blockchain entsprechen wird diese verworfen. Wenn die Transaktion den Regeln des Kontobuchs (Blockchain) entspricht, wird sie zum einen, in einer Liste mit Transaktionen hinzugefügt, die zukünftig in eine neue Kontobuch-Seite

eingearbeitet werden. Und zu anderen sendet der Empfänger die Transaktion an alle mit ihm verbundenen Teilnehmer, mit Ausnahme des Absenders, weiter. Somit verteilt sich eine Transaktion über das gesamte Netzwerk.

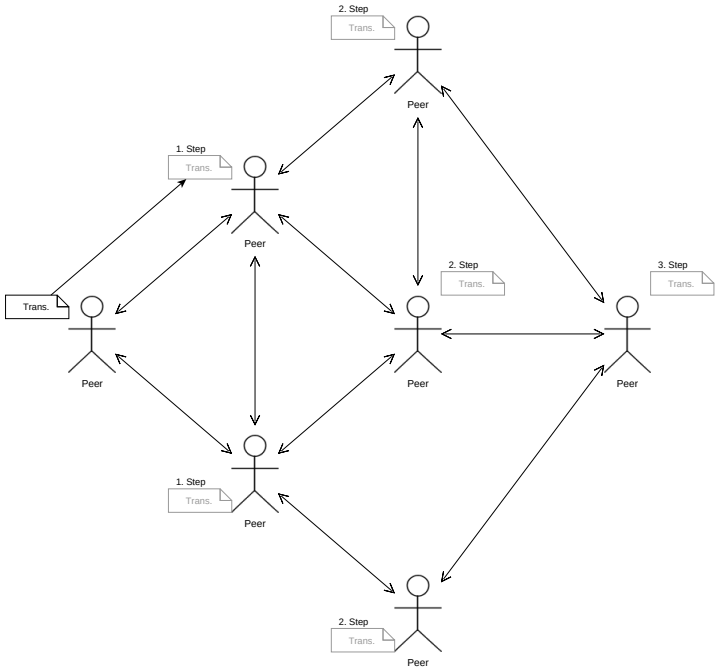


Abbildung 1.3: *Verteilung von Transaktion in einem Peer-to-Peer Netzwerk*

Die Regeln des Kontobuchs sehen dabei mindestens vor, dass die Transaktionen von den Auftraggeberkonten (Inhabern) unterschrieben und die Beträge gedeckt sind, also das zu transferierende Kapital dem Auftraggeber zur Verfügung steht und somit nicht schon in einer anderen Transaktion "ausgegeben" wurde. Das Mehrfache "ausgeben" desselben Geldes wird als double-spending bezeichnet. Das Ziel einer Blockchain ist es double-spending zu verhindern.

Eine Transaktion ist erst valide, wenn diese in einer Kontobuch-Seite eingepflegt wurde und diese Kontobuch-Seite von dem Großteil (>50%) der Kontobuch-Teilnehmer akzeptiert wurde, also zu dessen Kontobüchern hinzugefügt wurde.

Um zu gewährleisten das jeder Teilnehmer die Gültigkeit einer Seite des Kontobuchs überprüfen kann, enthält jede Seite des Kontobuchs einen Beleg. Dieser Beleg ist ein essenzieller Bestandteil einer Seite und ist in den meisten Blockchain-Implementierungen ein Arbeitsnachweis (man spricht hier im Englischen von Proof-of-work; PoW), also der Nachweis darüber einen gewissen Arbeitsaufwand beim Erstellen dieser Seite (Blocks) erfüllt zu haben.

Ein anderer Ansatz ist der Nachweis als Fertigsteller der Seite einen gewissen

Vermögensanteil (im Englischen Proof-of-stake; PoS) am gesamten Kapital im Kontobuch (zum aktuellen Stand des Kontobuchs) zu besitzen bzw. einen Arbeitsnachweis, mit einem Arbeitsaufwand, der umgekehrt proportional zum Gesamtvermögensanteil ist, zu erstellen.

Um Teilnehmer dazu zu bewegen den Aufwand, der beim Fertigstellen einer Seite bzw. der beim Erzeugen des Nachweises entsteht, auf sich zu nehmen, wird der Fertigsteller einer Seite belohnt. Diese Belohnung besteht zum einen aus der Summe aller Transaktionsgebühren, die auf der fertigzustellenden Seite anfallen und zum anderen aus einer (über die jeweiligen Regeln des Kontobuchs festgelegten) Vergütung für das Fertigstellen einer Seite des Kontobuchs.

Da die Vergütung, die für das Fertigstellen einer Seite des Kontobuchs gezahlt wird, nicht von Teilnehmern des Kontobuchs abgezogen wird, entspricht diese Vergütung neu erzeugtem Kapital die das Gesamtkapital im Kontobuch erhöht. Daher sehen, die meisten Regeln der Kontobücher (Blockchains) eine kontinuierliche (meist logarithmische) Abnahme dieses Teiles der Vergütung vor (die Vergütung in Form von Transaktionsgebühren bleiben bestehen). Dies soll das Gesamtkapital im Kontobuch auf einen Maximalbetrag beschränken und verhindert eine Inflation (bringt aber die Gefahr einer Deflation).

Es gibt noch andere Mechanismen zum Stabilisieren des Gesamtkapitals, das entscheidende Ziel ist aber immer das Gesamtkapital nach einer festgelegten Seitenzahl (Blocknummer) quasi stabil zu halten. Quasi, dadurch da bei einer rein logarithmischen Abhängigkeit eine Stabilität nie bzw. unter der Bedingung eines endlichen Wertebereiches erst sehr spät[1] eintritt.

Dem Fertigsteller einer Seite (Block) ist es erlaubt sich selbst die (ihm zustehende) Belohnung, für das Fertigstellen der Seite (Block), auszuzahlen. Dies geschieht, indem er zu der Seite (Block) die er fertigstellt, eine Transaktion hinzufügt, in der er sich selbst den Betrag der Belohnung gutschreibt.
Das Arbeiten an neuen Seiten (Blöcken), um entlohnt zu werden, wird als Mining (aus dem Englischen für Abbauen oder Fördern) bezeichnet. Teilnehmer, die dies tun, werden als Miner gezeichnet.

Da jeder Miner sich selbst die Belohnung gutschreiben möchte, ist die Transaktion, in der dies geschieht, bei jedem Teilnehmer anders. Somit unterscheidet sich die fertigzustellende Seite (Block) mindestens in dieser einen Transaktion, was dazu führt, dass die Basis des zu erstellenden Nachweises sich auch von Miner zu Miner unterscheidet. Was sich (unabhängig von der eingebrachten Arbeitsleistung) in unterschiedlichen Zeiten bis zu einem erfolgreichen Fertigstellen eines Nachweises auswirkt.

Da es jede Seitennummer (Blocknummer) nur ein einziges Mal in einem gültigen Kontobuch (Blockchain) geben kann, ist jeder Miner bestrebt eine neue Seite zu erstellen, die die nächste Seite zu der ihm bekannten höchsten Seite in seinem Kontobuch darstellt. Da die neue Seite (Block) auf die vorherige Seite (Block) verweisen muss, ist es ihm auch nicht möglich eine Seite mit einer noch größeren Seitennummer zu erstellen.

Schaft es ein Miner eine neue Seite fertigzustellen bevor dies ein anderer Miner (und

[1] Abhängig von der Größe des Wertebereiches

somit Teilnehmer) tut, bzw. bevor der Miner eine neue valide Seite mit der aktuellen Seitennummer empfängt, sendet der Miner die von ihm neu erstellte Seite an alle ihm bekannten Teilnehmer. Diese wiederum überprüfen ob die Seite (Block) den Regeln des Kontobuches (Blockchain) entsprechen und senden sie gegebenenfalls an die jeweils verbundenen Teilnehmer weiter. Somit verbreitet sich eine neue Seite im gesamten Netzwerk.

Sollte der Fall eintreten das zwei oder mehrere Miner nahezu gleichzeitig eine neue Seite fertiggestellt haben und diese sich im Netzwerk verbreiten, wird vorerst die jeweils zuerst erhaltene Seite (Block) zum Kontobuch (Blockchain) hinzugefügt und die andere(n) Seite(n) zur Seite gelegt.

Jeder Teilnehmer, der eine neue valide Seite empfängt, hebt diese grundsätzlich auf. Damit entsteht, aus den verketteten Seiten Abzweigungen die eine Baumstruktur bilden (siehe Abbildung 1.4). Die Wurzel der Baumstruktur bildet die erste Seite und von dieser aus wächst die Baumstruktur seitenweise in die Höhe. Dabei bilden sich bei Seiten mit gleicher Seitennummer Verästlungen aus. Die längste Kette, also die Kette aus Seiten mit der höchsten Seitennummer am Ende, ist die Hauptkette (englisch: main chain) und kann als Stamm der Baumstruktur gesehen werden. Die Hauptkette ist die Verkettung aus Seiten, die im Kontobuch liegen und somit die eine Kette, die gültig ist. Was eine Grundlage ist, um double-spending zu verhindern. Denn in einem Nebenast könnte das gleiche Geld in einer Transaktion, zu einem anderen Teilnehmer (als in der Hauptkette) transferiert worden sein.

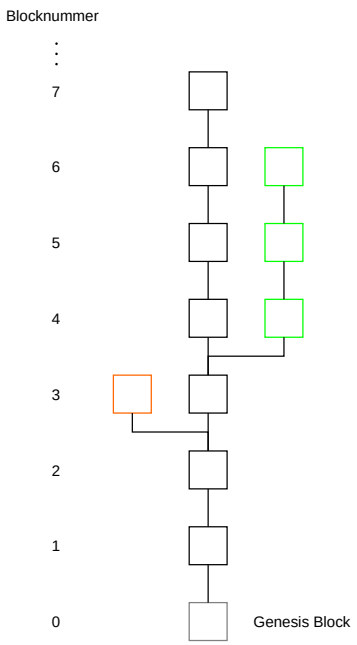


Abbildung 1.4: *Blockchain: Verkettung von Blöcken; Schwarze Blöcke bilden die gültige Hauptkette; Farbige Blöcke die Neben-Ketten*

Der für eine Seite zu erbringende Nachweis ist abhängig von einem Schwierigkeitsgrad, den der Nachweis mindestens erfüllen muss, um gültig zu sein. Dieser Schwierigkeitsgrad wird bei jeder neuen Seite neu errechnet und ist abhängig von den Schwierigkeitsgraden der letzten Seiten (Blöcke) und deren Fertigstellungszeiten und regelt somit, dass ein festes (im Kontobuch / in der Blockchain definiertes) Zeitliches-Intervall eingehalten wird, in denen neue Seiten (Blöcke) fertiggestellt werden können.

Dieses Intervall ist stark abhängig von der Blockchain (Beispiele siehe: Tabelle 1.1).

1.2.2 Proof-of-work

Proof-of-work (PoW) beschreibt im Zusammenhang mit der Blockchain-Technologie ein Algorithmus, bei dem aus einer Datenmenge (bei der Blockchain der Block-Header) ein Nachweis in Form eines Wertes (Codes) erstellt wird. Dieser Nachweis ist nur durch

einen sehr hohen Rechenaufwand erstellbar, aber umgekehrt leicht (mit sehr wenig Rechenaufwand) überprüfbar. Somit kann zum einen gesagt werden, dass ein (im Mittel) bestimmter Arbeitsaufwand aufgebracht werden musste, um den Nachweis zu erstellen. Und zum anderen kann sehr schnell überprüft werden, ob der erstellte Nachweis korrekt ist.

Implementiert wird das PoW Verfahren, indem der Datenmenge ein variabler Wert (so genanter Nonce) hinzugefügt wird. Der Nonce wird dabei so lange verändert, bis der Hash-Wert aus der Datenmenge und dem Nonce einen festgelegten Schwellwert unterschreitet. Der Schwellwert ist dabei umgekehrt proportional zum Schwierigkeitsgrad bzw. dem im Durchschnitt aufzuwendenden Rechenaufwand. Als Nachweis zählt ein Hash-Wert, der unterhalb des Schwellwertes liegt.

Bitcoin	10min [25]
Ethereum	15s [5]
Litecoin	2,5min [6]
Nxt	1min [7]

Tabelle 1.1: *Zeitliche Intervale in denen neue Blöcke veröffentlicht werden*

Wie sichert dieses Verfahren die Blockchain ab?

Bei einer Blockchain wird ein Intervall vorgegeben, in dem (im Durchschnitt) ein neuer Block erstellt wird (siehe Abbildung 1.1). Um dieses Intervall einzuhalten, wird der Schwierigkeitsgrad des PoW Nachweises bei jedem Block neu, an die aktuelle Gesamtrechenleistung des Blockchain-Netzwerks, angepasst. Die Gesamtrechenleistung des Blockchain-Netzwerks wird dabei aus den Schwierigkeitsgraden und den Zeitabständen der letzten X (abhängig von der Blockchain) Blöcke, geschätzt bzw. errechnet. Soll heißen, wenn der Block-Intervall unter dem Sollwert liegt, wird der Schwierigkeitsgrad sollange blockweise erhöht bis der Block-Intervall denn Sollwert erreicht.

Daraus folgt, dass es für einen Angreifer nur möglich ist, eine manipulierte längste (gültige) Kette zu erzeugen, wenn der Angreifer mehr Rechenleistung als die übrigen Teilnehmer (bzw. Miner) der Blockchain besitzt, also mehr als 50% der Gesamtrechenleistung des Blockchain-Netzwerks (inclusive des Angreifers).

1.2.3 Smart Contracts

Neben reinen Transaktionen (in denen einfach nur eine festgelegte Menge kryptografischem Geld von A nach B transferiert wird) gibt es die Möglichkeit sogenannte *Smart Contracts* zu erstellen.

Mit *Smart Contracts* werden im Allgemeinen Ablaufschemata bezeichnet, die Verträge in der realen Welt abbilden sollen. Diese Smart Contracts können in den verschiedenen Blockchain Implementierungen unterschiedliche Komplexitäten annehmen. So ist es zum Beispiel bei Bitcoin nur möglich einfache Contracts zu implementieren, die den Zugriff auf eine festgelegte Menge an kryptografischem Geld unter einer gewissen Voraussetzung gewährt[9]. Während zum Beispiel Ethereum komplexere und eingenständig in der Blockchain laufende Contracts ermöglicht, die als eingenständige Programme und Teilnehmer der Blockchain, auch höheren Anforderungen gerecht werden[37] (siehe 2.2 *Contracts in Ethereum*).

1.2.4 Zusatzinformationen in der Blockchain

Zusätzlich zu den eigentlichen Transaktionen, die einen differenziellen Status ab Beginn des ersten Blocks (Genesis Block) beschreiben, gibt es Blockchains wie z.B. Ethereum[37], die in jedem Block, aktuelle Statusinformationen mitführen. Diese Statusinformationen beschreiben den Zustand der Blockchain nach dem ausführen alle bisherigen Transaktionen. In Ethereum werden diese Status-Informationen als *World State* bezeichnet.

1.2.5 Vor- und Nachteile der Blockchain-Technologie

Die Blockchain-Technologie bittet einige Vor- und Nachteile gegenüber einer zentralen Datenbank. Diese Vor- und Nachteile haben zumeist mit der Dezentralität (Daten liegen verteilt bzw. redundant auf mehreren Knoten), den kryptografischen Eigenschaften der Blockchain oder mit dem der Blockchain zugrunde liegenden Meshnetzwerk (P2P-Protokoll) zu tun.

Auch werden die Schutzziele der Informationssicherheit, durch die Blockchain zum Großteil schon ohne zusätzlichen Aufwand gewährleistet [15]. Nur Vertraulichkeit muss durch zusätzliche Verschlüsselung (relevanter Daten) gewährleistet werden, da die Blockchain als offener Transaktionslog, global zur Verfügung steht (Transparenz).

1.2.5.1 Vorteile

- Dezentralisiert
- Unveränderlich
- Globales Plug & Play
- Unabhängig von einzelnen (Hersteller-) Services
- Vertrauens Aspekt (*Open Data*, globale Nachvollziehbarkeit)
- Extrem Anpassungsfähig - Auch der Ausfall mehrerer Knoten kann kompensiert werden.
- Sehr gute Skalierbarkeit - Keinen zusätzlichen Aufwand um eine Infrastruktur an das wachsende Blockchain-Netzwerk anzupassen.
- Extrem Robust - Robust gegen Angriffe und physikalischen (Hardware) Schaden bzw. Ausfall, da sich Netzwerk immer wieder anpasst.
- Verfügbarkeit - Verfügbarkeit insgesamt besser als bei dedizierten Architekturen.
- Angriffsresistent - Attacken auf einzelne Knoten im Netzwerk und selbst deren Korrumpierung ist kein Problem für die Integrität der gesamten Blockchain.
- Kosten - Reduktion oder Vermeidung von Kosten in der IT-Infrastruktur (intern wie auch extern).

1.2.5.2 Nachteile

- Energieverbrauch - Ein großer Nachteil einer Blockchain mit Prof-of-Work Implementierung ist der enorme Energieverbrauch der aus der verbunden benötigten Rechenleistung resultiert.
- Geringer Transaktionsdurchsatz
- Einschränkung im Speicherplatz
- (Vereinzelt) hohe Transaktionskosten (Bitcoin ≈0,58€) - Für größere Finanztransaktionen recht günstig, doch für kleinere Interaktionen mit der Blockchain, wie zum Beispiel das ändern einer Berechtigung, relativ kostenintensiv.

Zu beachten ist dabei, dass sich die ergebenden Nachteile durch den Einsatz von aktuellen Blockchains (die auf Finanztransaktionen zugeschnitten sind) und nicht aus der Blockchain-Technologie selbst ergeben.

1.3 Triple-A-Systeme

Triple-A-Systeme (AAA-Systeme) sind Systeme zur Authentifizierung, Autorisierung und Abrechnung (im Englischen: Authentication, Authorization and Accounting) und werden für den Netzwerkzugang zu sowohl kabelgebundenen wie auch drahtlosen Netzen eingesetzt [16].

Damit werden diese zum Großteil von Internetprovidern eingesetzt, aber auch von größeren Institutionen, öffentlichen Einrichtungen (wie Universitäten, Schulen, usw.) oder Unternehmen, die AAA-Systeme für deren WLAN-Netzwerke einsetzen (wenn auch im kleineren Maßstab und möglicherweise nur einem *Accounting* in Form eines Protokolls der Nutzeraktivität).

Zur Authentifikation des Nutzers bzw. des Netzwerk-Clients bieten sich Protokolle wie das *Extensible Authentication Protocol* (EAP) an.

1.4 Extensible Authentication Protocol (EAP)

Das *Extensible Authentication Protocol* (EAP; deutsch: Erweiterbares Authentifizierungs-Protokoll) ist ein, von der *Internet Engineering Task Force* (IETF) unter *RFC3748* [8] veröffentlichtes erweiterbares Protokoll oder genauer, ein Authentifizierungs-Framework das viele verschiedene Authentifizierungsmethoden (EAP-Methoden) unterstützt [8].

EAP wurde entwickelt, um die Authentifizierung für einen Netzwerk-Zugriff durchzuführen. Somit ist EAP auch für den Fall ausgelegt, dass (noch) keine IP-Schicht zur Verfügung steht. Typischerweise sitzt EAP, im OSI-Model (aus dem Englischen: Open Systems Interconnection Model), direkt oberhalb der Sicherungsschicht (Data Link Layer; wie das Point-to-Point Protocol (PPP) oder IEEE 802), ohne dass das IP zum Einsatz

kommt. Daher implementiert EAP selbst Paketverwaltungsstrategien, wie zum Beispiel das Entfernen doppelter Pakete oder das automatische erneute Senden eines Paketes. Die automatische Aufteilung von großen Datenmengen auf mehrere Pakete (im Englischen: Fragmentation) wird von EAP selbst nicht unterstützt [8]. Das heißt, wenn eine EAP-Methode Nutzdaten erzeugen, die größer sind als die minimale Größe der EAP MTU (Maximum Transmission Unit; im Deutschen: maximale Übertragungseinheit), muss diese EAP-Methode sich selbst um eine Fragmentation deren Nutzdaten kümmern.

Dabei ist, in dem EAP-Standard festgelegt, dass die darunter liegende Netzwerkschicht eine MTU-Größe von mindestens 1020 Bytes oder größer unterstützen muss [8].

Eine Authentifizierung über EAP wird (im Gegensatz zu vielen anderen Authentifizierungsprotokollen) immer von dem Server, dem sogenannten Authenticator (im Deutschen: Authentifizierer) eingeleitet.

EAP ist nach dem Request-Response- (im Deutschen: Anfrage-Antwort) Prinzip entworfen worden. Der Authenticator ist dabei immer derjenige der Anfragen, in Form von Request-Nachrichten, an den Peer stellt. Der Peer beantwortet diese Anfragen durch Response-Nachrichten. Jede Request- und Response-Nachricht ist auf ein Paket beschränkt und es kann nur eine Nachricht (pro Peer) gleichzeitig unterwegs sein, was heißt das (pro Peer) immer nur eine Anfrage bis zur erhalten der Antwort versendet werden kann. Somit ist es pro Umlauf, der aus einer Anfrage und einer Antwort besteht, nur möglich (in jede Richtung) Daten mit einer maximalen Paketgröße in der Größe der minimalen EAP MTU zu versenden.

Diese Einschränkungen sind dem einfachen Aufbau eines EAP-Pakets geschuldet (siehe Abbildung 1.5).

1.4.1 Paketaufbau

Abbildung 1.5: *EAP-Protokoll-Paketaufbau [8]*

Beschreibung der Abbildung 1.5 [8]:

Code
Typ des EAP-Pakets:

1 Request

2 Response

3 Success

4 Failure

Identifier
1 Byte lange Identifizierungsnummer um eine Antwort einer Anfrage zuzuordnen

Length
Länge (in Bytes) des gesamten EAP-Pakets (also Länge der Felder: Code, Identifier, Length und Data), somit immer größer gleich 4

Data
Nutzdaten, ≥ 0 Bytes, Format abhängig vom Code des Pakets

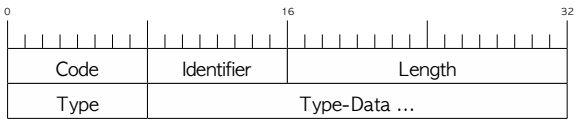


Abbildung 1.6: *EAP-Protokoll - Request- und Response-Paketaufbau [8]*

Beschreibung der Abbildung 1.6:

Code
Typ des EAP-Paketes bzw. der EAP-Methode:

1 Request

2 Response

Identifier
siehe Beschreibung der Abbildung 1.5

Length
siehe Beschreibung der Abbildung 1.5

Type
Methoden-Typ, Type des Request- oder Response-Pakets bzw. der EAP-Methode

Type-Data
Nutzdaten, ≥ 0 Bytes, Format abhängig von der EAP-Methode bzw. dem Methoden-Typ und dem Code (Request oder Response)

Übersicht grundlegender (bzw. initialer) Methoden-Typen (Request-/Response-Typen) [8]:

1 Identity

2 Notification

3 Nak (nur bei Response-Paketen)

4 MD5-Challenge

5 One Time Password (OTP)

6 Generic Token Card (GTC)

254 Expanded Types

255 Experimental use

1.4.2 Expanded Types

Um die, seit den Anfängen von EAP immer mehr gewordene, anbieterspezifische Nutzung von EAP zu unterstützen, wurden das Type-Feld (dass hauptsächlich in den Request- und Response-Paketen verwendet wird, siehe Abbildung 1.6) erweitert.

Der sogenannten *Expanded Type* soll dabei den globale Methoden-Typ Bereich, von den ursprünglichen 255 möglichen Typen (angegeben durch das 8-Bit Type-Feld), erweitern.

Der *Expanded Type* besteht dabei neben dem ursprünglichen Feld *Type*, der mit dem Wert 254 die *Expanded Types*-Erweiterung signalisiert, aus zwei weiteren Feldern: *Vendor-Id* und *Vendor-Type* (siehe Abbildung 1.7).

Das (3 Byte lange) Feld *Vendor-Id* gibt die Identifizierungsnummer des Anbieters an. Die Identifizierungsnummer repräsentiert dabei den *SMI Network Management Private Enterprise Code* des Anbieters, welcher von der *Internet Assigned Numbers Authority* (IANA) vergeben wird. Die Identifizierungsnummer 0 ist dabei für die IETF reserviert.

Das (4 Byte lange) Feld *Vendor-Type* gibt einen anbieterspezifischen Methoden-Typ an.

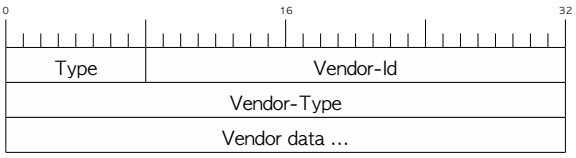


Abbildung 1.7: *EAP-Protokoll-Paketaufbau: Typen-Feld Erweiterung [8]*

Beschreibung der Abbildung 1.7 [8]:

Type
= 254 für *Expanded Type*

Vendor-Id
Identifizierungsnummer des Anbieters der EAP-Methode

Vendor-Type
Anbieterspezifischer Typ der EAP-Methode

Vendor-Data
Nutzdaten, $\geq$ 0 Bytes, Format abhängig von der EAP-Methode bzw. dem Methoden-Typ und dem Code (Request oder Response)

1.4.3 Sicherheitsbetrachtungen

Um ganz klar verständlich zu machen welche Sicherheit eine EAP-Methode bereitgestellt, muss jede Spezifikation einer EAP-Methode über einen Abschnitt mit Sicherheitsansprüchen verfügen [8].

Die Deklarationen was in diesem Abschnitt mit Sicherheitsansprüchen enthalten sein muss, ist im Kapitel 7.2 *Security Claims* des EAP-Standards beschrieben (RFC3748; siehe Anhang A.2).

Weitere (allgemeine) Informationen zu Thema Sicherheit sind im Kapitel 7 *Security Considerations* des EAP-Standards (RFC3748 [8]) zu finden.

1.5 IEEE 802.1X

IEEE 802.1X ist ein Standard für eine anschlussbasierende Netzwerkzugangskontrolle für Lokale- und Großraum-Netzwerke [34]. Und wird häufig von Internetprovidern mit deren AAA-Systemen eingesetzt, aber auch in großen WLAN-Netzen (von Unternehmen, usw.) zur Nutzerauthentifizierung in Kombination mit EAP eingesetzt.

Die Abbildung 1.8 zeigt die Schnittstelle zwischen *IEEE 802.1X* und EAP.

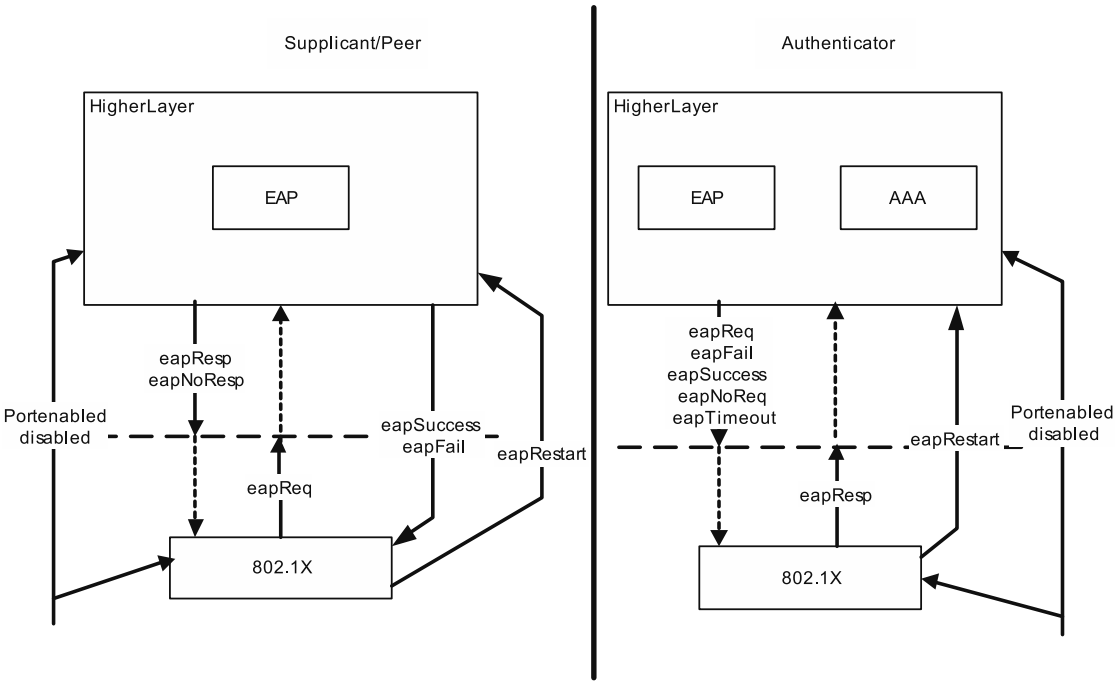


Abbildung 1.8: *Higher layer interface diagram (IEEE Std 802.1X-2004: Fig. 8-1 [34])*

In Abbildung 1.9 ist ein Sequenzdiagramm gezeigt, dass eine Authentifizierung für ein WLAN-Netzwerk nach dem *IEEE 802.11*-Standard zeigt. Dabei ist zu sehen wie EAP-Datenpakete nach dem *IEEE 802.1X*-Standard zwischen dem Nutzergerät (Supplicant) und dem Authenticator (der dem WLAN-Access-Point entspricht) ausgetauscht werden [35].

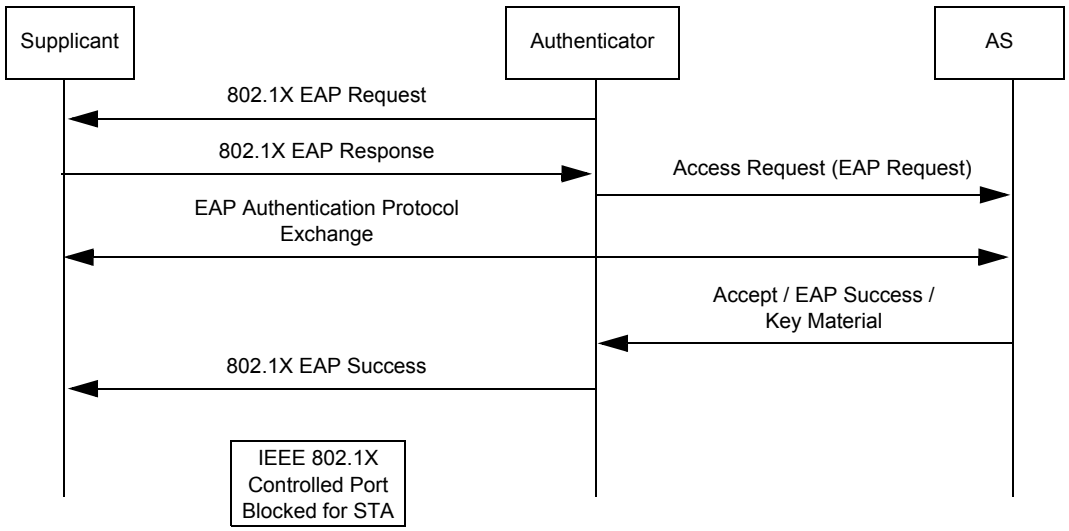


Abbildung 1.9: *IEEE 802.1X EAP authentication (IEEE Std 802.11-2012: Fig. 4-18 [35])*

Kapitel 2

Voruntersuchung

2.1 Blockchain

Der aktuelle Großteil aller Blockchains ist für den Einsatz im Finanzsektor bzw. auf Finanztransaktionen zugeschnitten. Nur wenige Blockchains gehen mit ihrer Funktionalität in eine andere Richtung.

Ausnahmen sind hier Blockchain-basierte Datenbanken (wie beispielsweise BigchainDB [22]) und DNS-ähnliche Systeme (wie beispielsweise Namecoin [3]).

Auch gibt es das Projekt Hyperledger, das Werkzeuge zur Verfügung stellt, um eigene (auf Anwendungen zugeschnittene) Blockchains zu erstellen [2].

Des Weiteren bieten Blockchains, ab der zweiten Generation (erste Generation stellt Bitcoin dar), immer mehr Funktionalität im Bereich der Smart-Contracts. Diese Smart-Contracts ermöglichen es benutzerdefinierte Daten in der Blockchain abzulegen und in dieser zu verarbeiten.

Damit erfüllen Blockchains, die Smart-Contracts unterstützen, eine wichtige Voraussetzung um als Backend eines Triple-A-Systems infrage zu kommen.

Eine Eigene, auf das Backend eines Triple-A-Systems zugeschnittene, Blockchain ist immer die beste Lösung, da sie auf alle nur machbaren Anforderungen hin entwickelt werden kann. Möchte man keine eigene, auf die Anforderungen zugeschnittene, Blockchain entwickeln, sind Blockchains die Smart-Contracts unterstützen eine gute Alternative.

Die zusätzliche Entwicklungszeit einer eigenen Blockchain würde denn zeitlichen Rahmen dieser Masterarbeit sprengen. Somit ist eine Blockchain, die Smart-Contracts unterstützt, die optimale Lösung aus zeitlicher Machbarkeit und der Erfüllung der Anforderungen.

2.1.1 Nähere Betrachtung verschiedener Blockchains

Hier werden einige Blockchains, auf die Anwendbarkeit in dieser Arbeit, näher untersucht.

2.1.1.1 Bitcoin

Bitcoin ist die Erste und somit älteste Blockchain. Und durch viele Informationen und Dokumente rund um Bitcoin, ist diese auch schnell und einfach nutzbar. Aber Bitcoin bittet nicht die Möglichkeit eigene (benutzerspezifische) Daten in der Blockchain abzulegen. Somit ist Bitcoin für den Anwendungsfall in dieser Arbeit nicht geeignet [25].

2.1.1.2 Hyperledger

Zum Erstellungszeitpunkt dieser Arbeit befindet sich das Hyperledger Projekt noch in einer sehr frühen Phase. Dadurch ist, bei dessen Einsatz, mit einem erhöhten Zeitaufwand zu rechnen. Der erhöhte zeitliche Aufwand lässt sich unter anderem damit begründen, dass es bei Hyperledger nicht um eine Blockchain an sich handelt, sondern nur als Plattform oder Werkzeugsammlung, zum Erstellen eigener Blockchains, gesehen werden kann [2].

Dies ist zwar für den Anwendungsfall in dieser Arbeit von Vorteil, da eine Blockchain erstellt werden kann, die allen (machbaren) Anforderungen dieses Projektes gerecht wird. Aber dies bedeute auch, dass diese Blockchain samt Infrastruktur, erst zeit- und ressourcenintensiv erstellt werden muss.

Was im Endeffekt einen wesentlich erhöhten Arbeitsaufwand bedeutet, der im Gegensatz zu dem einfachen Verwenden einer fertigen Blockchain bzw. Blockchain-Infrastruktur, wie Bitcoin oder Ethereum steht. Somit ist Hyperledger trotz seines Potenzials für diesen Anwendungsfall, aufgrund des eingeschränkten Zeitrahmens dieser Masterarbeit, leider auszuschließen.

2.1.1.3 Ethereum

Ethereum ist eine Blockchain der zweiten Generation, die speziell für den Einsatz von Smart-Contracts entwickelt wurde [37]. Diese Smart-Contracts ermöglichen es Nutzern nicht nur eigene (kleine) Programme in der Blockchain laufen zu lassen, sondern ermöglichen es auch individuelle Daten an, in der Blockchain liegende, Smart-Contracts zu senden und somit die Daten in der Blockchain zu hinterlegen (speichern). Damit erfüllt Ethereum nicht nur eine wichtige Voraussetzung für den Einsatz in dieser Arbeit (das benutzerdefinierte Daten speicherbar sind), sondern ermöglicht es auch Funktionalität bzw. Methoden des Triple-A-Systems wie zum Beispiel die Autorisierung direkt in einen Smart-Contract zu implementieren.

Dies macht Ethereum zur optimalen Blockchain für diese Arbeit.

Angemerkt sei noch, dass die offizielle Währung in Ethereum der sogenannte *Ether* (ETH) ist, ein *Ether* entspricht dabei dem 10^{18} fachen der kleinsten möglichen Währungseinheit *Wei*. Weitere Währungseinheiten sind der *Szabo* der dem 10^{12} fachen eines *Wei* entspricht und der *Finney* der dem 10^{15} fachen eines *Wei* entspricht [37]. Laut der Ethereum-Spezifikation (*2.1. Value*) ist für *Ether* manchmal auch das altenglische Ð gebräuchlich.

2.2 Contracts in Ethereum

Smart-Contracts werden in Ethereum verkürzt nur als Contracts bezeichnet, sie verfügen über eine Ethereum-Adresse (können also Transaktionen empfangen und senden) und über ausführbaren Bytecode.

Der Bytecode des Contracts wird beim erstellen des Contracts mit übergeben und beschreibt das komplette Verhalten und die Funktionalität des Contracts.

Alle Kosten die ein Contract produziert, werden in so genantem *Gas* berechnet. Dieses *Gas* kann in *Ether* (der Standart Währung in Ethereum) umgerechnet werden. Wobei der Kurs *Gas* - *Ether* sich immer, abhängig von der aktuellen Belastung der Miner, von Block zu Block ändern kann.

Contracts werden bevorzugt in der Programmiersprache Solidity geschrieben bzw. programmiert und dann in Bytecode kompiliert (bevorzugt, da es nur für Solidity einen gut funktionierenden Compiler gibt).

Jeder ausgeführte Befehl des Contract-Bytecodes kostet eine festgelegte Menge *Gas* (siehe *Appendix G. Fee Schedule* in der Ethereum-Paper *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER* [37]) und addiert sich über denn kompletten Aufruf (einer Methode) des Contracts, zu eine Summe auf. Diese *Gas*-Kosten werden bei der Transaktion im Vorfeld durch einen Maximalen-*Gas*-Betrag abgedeckt. Der Maximale-*Gas*-Betrag (der bei der Transaktion angegeben wird) vermeidet somit auch, dass ein Contract über eine eine sehr lange Zeit oder Endlos Ausgeführt wird, da der Bytecode-Interpreter die Codeausführung abbricht wenn der maximale angegebene *Gas*-Betrag erschöpft ist.

Der Contract kann (dauerhaften) Speicher (innerhalb des World-States der Blockchain) für sich reservieren bzw. nutzen. Das speichern solcher dauerhaften Informationen kostet den Contract eine festgelegte Menge *Gas* (20000 *Gas* für ein 32 Byte Speicherplatz [37]).

2.3 Micropayment

Das Ziel bei Micropayments (Kleinbetragszahlungen) ist das kostengünstige transferieren kleiner und sehr kleiner Beträge [18]. Dazu werden möglichst viele Micropayment-Transaktionen zu einer Transaktion in der Blockchain gebündelt. Die Liste (oder aus dem englischen 'channel'; Kanal) in der alle Micropayment-Transaktionen zu einer Transaktion gebündelt werden, wird als Micropayment-Channel bezeichnet. Dabei fallen idealerweise nur die Transaktionsgebühr für die eine (gebündelte) Transaktion, die beim schliessen des Micropayment-Channel an die Blockchain gesendet wird, an.

Bei der Bündelung der Micropayment-Transaktionen wird nur die Summe (oder Summen bei mehreren Absender- und/oder Empfängern-Konten) aller Micropayment-Transaktionen in die abschliessende Transaktion gepackt. Je mehr Micropayment-

Transaktionen zu einer Transaktion gebündelt werden, je günstiger wird jede einzelne Micropayment-Transaktion, da die eine Transaktionsgebühr der abschliessenden Transaktion auf alle Micropayment-Transaktionen verteilt gesehen werden kann.

Die Herausforderung ist Sicherzustellen, dass der oder die Empfänger nach Beendigung einer gewissen Bedingung zum Beispiel einer festgelegten Zeit oder einer bestimmten Menge von Micropayment-Transaktionen, ihre aktuelles Bilanz-Guthaben des Micropayment-Channels (Summe ihrer Micropayment-Transaktionen) erhalten. Um sicherzustellen, dass beim Abschluss des Micropayment-Channels beim Absender der Micropayments die erforderliche Summe auch zur verfügung steht, legt der Absender einen Maximalbetrag für einen Micropayment-Channel fest. Dieser Maximalbetrag wird dann beim erstellen des Micropayment-Channel vom Guthaben des Absenders gesperrt. Wie diese Sperrung von Guthaben in der Praxis aussieht ist abhängig von der jeweiligen verwendeten Blockchain.

In einem einfachen Micropayment-Channel sind zwei Parteien involviert. Es können aber auch komplexere Micropayment-Channels gebildet werden, die aber zumeist auch nur aus, komplex miteinander verwoben, zweiparteigen Micropayment-Channel bestehen.

Je nach verwendeter Blockchain ist ein Micropayment-Channel unterschiedlich implementiert.

In dieser Arbeit ist Micropayment ein wichtiges Mittel um kleine Daten-Volumen- oder Zeit-Pakete wirtschaftlich Abrechnen zu können.

2.3.1 Micropayment-Contract

In Ethereum ist ein Smart-Contract das Mittel der Wahl zur Implementierung eines Micropayment-Channels. Dieser Micropayment-Contract wird vom Initiator des Micropayments erstellt. In dem nachfolgend gezeigten Micropayment-Contract ist der Initiator auch immer der Absender der Transaktionen. Dies ist im Fall eines Micropayment-Contracts auch sinnvoll da der Absender in jedem Fall einen gewissen Betrag, der dem Maximalbetrag für die Micropayments entspricht, zurücklegen muss. Mit zurücklegen ist in diesem Fall das Sicherstellen eines gewissen Betrages gemeint, sodass der Empfänger die Sicherheit hat, dass dieser Betrag auch beim Abschluss aller Micropayments zur Verfügung steht. Entspricht nun der Initiator auch dem Absender, lässt sich beim Erstellen das Micropayment-Contracts auch gleichzeitig, also in einer Transaktion, der Smart-Contract erstellen und der Micropayment Maximalbetrag transferieren. Somit ist nur für das Erstellen des Smart-Contract eine Transaktionsgebühr (eng. transaction fee) zu entrichten und keine weitere mehr für das Transferieren des Micropayment-Maximalbetrags.

Nachfolgend ist die Funktionsweise als Beispiel gezeigt:

Bob möchte Alice für die Nutzung ihres WLAN-Netzes eine Vergütung zukommen lassen.

Der Datenverkehr (Traffic) soll dabei auf ein Datenpaket genau abgerechnet werden.

Die Größe eines Datenpaketes sei dabei auf 1 MB festgelegt, kann aber auch auf eine beliebig andere sinnvolle Größe festgelegt werden.

Da die Transaktionsgebühren in den bekanten Blockchains wie Bitcoin oder Ethereum die Kosten eines Paketes übersteigen, ist es hier sinnvoll einen Micropayment-Channel oder im Falle von Ethereum einen Micropayment-Contract zu verwenden. Dieser Micropayment-Channel ermöglicht es paketweise Abrechnen zu können, ohne bei jeder Vergütung eines Pakets eine Transaktionsgebühr zahlen zu müssen.

Der hierzu entwickelte Micropayment-Contract sieht nur das Zahlen von drei Transaktionsgebühren vor. Und erlaubten im Gegenzug beliebig oft, gebührenfreie Micropayments bis zu einem maximal festgelegten Betrag zu tätigen.

Um eine Transaktion vonseiten des Absenders zu verifizieren, übergibt der Absender dem Empfänger des Micropayments einen sogenannten Micropayment-Accesstoken. Dieser Micropayment-Accesstoken ist ein Code der die Micropayment-Transaktion von Seiten des Absenders bestätigt und mithilfe des Micropayment-Contracts überprüfbar bzw. einlösbar ist.

Der Micropayment-Contract ist dabei so aufgebaut, dass er als eigenständige dritte Partei arbeitet, also nach der Erstellung nicht mehr von einer der beiden anderen Seiten (Absender oder Empfänger der Micropayments) beeinflusst werden kann.

2.3.1.1 Micropayment-Contract per ECDSA

Bei einem Micropayment-Contract, der mit ECDSA-Signaturen arbeitet, wird bei dessen Erstellung (in der Blockchain) ein öffentlicher (ECDSA) Schlüssel hinterlegt, mit dessen Hilfe der Contract (in der Blockchain) die Gültigkeit von Micropayment-Accesstoken überprüfen kann. Die Schlüssel werden nachfolgend als öffentlicher und privater Micropayment-Schlüssel bezeichnet.

Die Micropayment-Accesstoken bestehen aus einem Access-Value und einer Access-Signature. Der Access-Value entspricht der Summe aller Micropayments vom Absender an den Empfänger in diesem Micropayment-Contract, also dem Betrag, den Alice beim Abschluss des Micropayment-Contracts erhält. Die Access-Signature ist eine ECDSA-Signatur die der Absender mithilfe des (nur ihm bekannten) privaten Micropayment-Schlüssels erstellt.

Die folgende Fortsetzung des Beispiels verdeutlicht die Funktionsweise eines Micropayment-Contract, der mit ECDSA-Signaturen arbeitet:

Bob möchte Alice Zugriff auf 0,001 ETH (Ether) aus seinem (für Alice erstelltem) Micropayment-Contract geben, um damit ein weiteres Datenpaket zu bezahlen. Dazu

erstellt Bob einen Micropayment-Accesstoken, bei dem der Access-Value auf den letzten Access-Value plus 0,001 ETH festgelegt wird. Bob gibt den erzeugten Micropayment-Accesstoken an Alice weiter. Diese kann den Micropayment-Accesstoken dann über eine Methode des Micropayment-Contracts validieren bzw. auf Funktionalität prüfen.

Dieser Vorgang kann beliebig oft wiederholt werden, wobei keine Kosten für die einzelnen Micropayments anfallen.

Zum Abschließen des Micropayment-Vorgangs (was vor Ablauf der Lebenszeit der Micropayment-Contracts passieren muss), kann Alice das letzte und somit das höchstwertige Micropayment-Accesstoken einlösen. Dies geschieht über eine Methode des Micropayment-Contracts, die in der Blockchain ausgeführt wird und Alice, die im Micropayment-Accesstoken angegebene Summe vom Guthaben (der Balance) des Micropayment-Contracts transferiert.

Nach Ablauf der Lebenszeit kann Bob über eine Methode des Micropayment-Contracts, den Micropayment Vorgang abschließen. Daraufhin löscht sich der Micropayment-Contract selbst und transferiert abschließend das restliche Guthaben (die restliche Balance des Micropayment-Contracts) zurück zu Bob.

2.4 Implementierungsumgebung

Um zu demonstrieren, dass die, in dieser Arbeit gezeigte Technologie praxistauglich ist, soll die bei der Implementierung eingesetzte Hardware- und Softwareplattform Ähnlichkeit mit der von handelsüblichen WLAN-Access-Points besitzen.

Was im Einzelnen heißt, dass das System auf einem eingebetteten System (im Englischen: Embedded System) laufen muss, bei dem häufig ein ARM-Prozessor im Einsatz ist und nur ein sehr begrenzter Arbeitsspeicher und Flashspeicher zur Verfügung steht. Die dabei am häufigsten eingesetzten Betriebssystem sind Linux-Systeme.

Die (zurzeit) naheliegendste Lösung ist somit ein *Raspberry Pi 3* System. Diese eingebetteten Systeme verfügen sowohl über eine Ethernet-Schnittstelle, wie über eine integrierte WLAN-Schnittstelle, was einen Einsatz als WLAN-Access-Point ermöglicht. Und stellen zusätzliche eine höhere Prozessorleistung und mehr Speicher, im Vergleich zu üblicher WLAN-Access-Point-Hardware, zur Verfügung. Dies lässt einen zusätzlichen Spielraum für Leistungseinbußen, die bei der Implementierung eines ersten Prototyps bzw. bei einer noch frühen (und somit noch nicht optimierter) Softwareversion auftreten können.

Des weitern ermöglicht die performantere Hardwareplattform auch den Einsatz eines normalen (nicht für leistungsschwache Embedded-Systems zugeschnittene) Blockchain-

Clienten.

2.5 Kryptosysteme

Dieser Abschnitt dient der Voruntersuchung geeigneter Kryptosysteme (Kryptoverfahren) für diese Arbeit. Dabei werden die aktuell als sicher geltenden Verfahren betrachtet, also Verfahren, die öffentlich und langfristig bekannte sind und dadurch schon von unzähligen Kryptologen auf Sicherheitsdefizite hin, analysiert worden sind.

2.5.1 Asymmetrisches Kryptosystem

Mit asymmetrischem Kryptosystem wird meist ein kryptografischer Algorithmus bezeichnet, der im Unterschied zu symmetrischen Algorithmen (die einen einzigen Schlüssel verwenden), zwei unterschiedliche Schlüssel zur Ver- und Entschlüsselung bzw. zum Erstellen und Überprüfen von digitalen Signaturen verwendet. Die Schlüssel, des Schlüsselpaares, werden auch öffentlicher Schlüssel (im Englischen: public key) und privater Schlüssel (im Englischen: private key) genannt. Daher auch der andere Name für asymmetrische Kryptosysteme: Public-Key-Kryptosysteme.

Zu den asymmetrischen Kryptoverfahren zählen auch Protokolle wie zum Beispiel der Diffie-Hellman-Schlüsselaustausch, der zugleich auch als das erste asymmetrische Kryptoverfahren gilt [14].

Der Diffie-Hellman-Schlüsselaustausch (DH) ist ein Protokoll, mit dem über einen unsicheren Kanal ein geheimer Schlüssel vereinbart werden kann.

Was asymmetrische Kryptosystem wie den DH-Schlüsselaustausch, in Kombination mit einem digitalen Signatur-Verfahren zur (gegenseitigen) Authentifizierung der Kommunikationspartner, unverzichtbar für die moderne Kryptografie macht.

Alle heutigen relevanten digitalen Signatur-Verfahren sind im *Digital Signature Standard* (DSS) festgehalten. Darunter sind [27]:

- **DSA** - Digital Signature Algorithm
- **RSA** - RSA digital signature algorithm
- **ECDSA** - Elliptic Curve Digital Signature Algorithm

2.5.1.1 Klassifizierung von asymmetrischen Kryptosystemen

Asymmetrische Kryptosysteme lassen sich anhand ihres Grundproblems differenzieren bzw. in Klassen einordnen.

Kryptosysteme, die auf dem Problem des diskreten Logarithmus basieren (im Englischen: Discrete Logarithm Cryptography; DLC), lassen sich in zwei Unterkategorien teilen, der Kryptografie in endlichen Körpern (im Englischen: Finite Field Cryptography; FFC; auch Galoiskörper Kryptografie) und der elliptischen Kurven Kryptografie (im Englischen: Elliptic Curve Cryptography; ECC). Dabei unterscheiden sich diese in der verwendeten Mathematik [27].
Zur DLC zählen DSA und der Diffie-Hellman-Schlüsseltausch, welche beide als FFC- oder ECC-Variante existieren.

RSA zählt zu den Kryptosystemen, die auf dem Problem der Primfaktorzerlegung basieren (im Englischen: Integer Factorization Cryptography; IFC) [27].

Zu den modernen Kryptosystemen zählt ECC.

2.5.1.2 Elliptische-Kurven-Kryptografie (ECC)

Elliptische-Kurven-Kryptografie (ECC) ist ein Oberbegriff für asymmetrische Kryptosysteme, deren Sicherheit auf dem Problem des diskreten Logarithmus in elliptischen Kurven (im Englischen: Elliptic Curve Discrete Logarithm Problem; ECDLP) basiert [20].

Formel der ECC-Kurve:

$$y^2 = x^3 + ax + b \tag{2.1}$$

Da das Problem des diskreten Logarithmus in elliptischen Kurven wesentlich schwieriger zu berechnen ist, als Problem des diskreten Logarithmus alleine, reichen bei ECC wesentlich kleinere Schlüssellängen aus, um die gleiche Sicherheit zu erzielen.

Verdeutlicht wird dies durch die Tabellen 2.1 und 2.2.

Aus der kleineren Schüssellängen resultieren auch Einsparungen in der jeweils benötigen Energie, Speicherbedarf, Bandbreite und Rechenleistung [10].

Auch lassen sich die älteren, auf dem diskreten Logarithmus in endlichen Körpern basierenden, Kryptosysteme wie DH und DSA zu ECC-Varianten umformen (ECDH und ECDSA), was deren Nutzung mit elliptischen Kurven vereinfacht.

Diese Vorteile erklären, warum mit ECC immer mehr ältere Kryptosysteme wie DH und DSA erweitert, oder im Fall von RSA ersetzt werden.

Üblicherweise werden für ECC, als endliche Körper (Galoiskörper) nur Primfelder (Primkörper; $\mathbb{F}_p = GF(p)$; im Englischen: Prime Field) oder Binäre-Felder ($\mathbb{F}_{2^m} = GF(2^m)$; im Englischen: Binary Field) verwendet [27] [20].

Die Tabelle 2.1 zeigt das Sicherheitsniveau (in Bit) im Vergleich mit der Schlüssellänge (in Bit) von Kryptosystemen wie RSA, DSA oder DH.

Sicherheitsniveau	RSA/DSA/DH
56	512
64	704
80	1024
96	1536
112	2048
128	3072
192	7680
256	15360

Tabelle 2.1: *Sicherheitsniveau verschiedener RSA/DSA/DH-Schlüssellängen [32]*

Die Tabelle 2.2 gibt eine Übersicht über verschiedene ECC-Kurven. Dabei ist dargestellt welches Sicherheitsniveau (in Bit) eine standardisierte Kurve mit ihren spezifischen Parametern besitzt. Auch ist angegeben ob die Kurve von der ANSI, empfohlen (im Englischen: recommended; in der Tabelle mit r gezeigt) oder mit dem Standard konform (im Englischen: conformant; in der Tabelle mit c gezeigt) ist. Die gezeigten Daten stammen aus dem Standard *SEC 2: Recommended Elliptic Curve Domain Parameters* der *Standards for Efficient Cryptography (SEC)* und die verwendeten Tabellen sind im Anhang A.1 zu finden [32] [10].

Sicherheitsniveau	$\mathbb{F}$	Schlüssellänge	Koblitz / Random	SEC Name	ANSI X9.62	ANSI X9.63	NIST Bezeichnung
56	$\mathbb{F}_p$	112	r	secp112r1	-	-	-
56	$\mathbb{F}_p$	112	r	secp112r2	-	-	-
56	$\mathbb{F}_{2^m}$	113	r	sect113r1	-	-	-
56	$\mathbb{F}_{2^m}$	113	r	sect113r2	-	-	-
64	$\mathbb{F}_p$	128	r	secp128r1	-	-	-
64	$\mathbb{F}_p$	128	r	secp128r2	-	-	-
64	$\mathbb{F}_{2^m}$	131	r	sect131r1	-	-	-
64	$\mathbb{F}_{2^m}$	131	r	sect131r2	-	-	-
80	$\mathbb{F}_p$	160	k	secp160k1	c	r	-
80	$\mathbb{F}_p$	160	r	secp160r1	c	c	-
80	$\mathbb{F}_p$	160	r	secp160r2	c	r	-
80	$\mathbb{F}_{2^m}$	163	k	sect163k1	c	r	NIST K-163
80	$\mathbb{F}_{2^m}$	163	r	sect163r1	c	c	-
80	$\mathbb{F}_{2^m}$	163	r	sect163r2	c	r	NIST B-163
96	$\mathbb{F}_p$	192	k	secp192k1	c	r	-
96	$\mathbb{F}_p$	192	r	secp192r1	r	r	NIST P-192
96	$\mathbb{F}_{2^m}$	193	r	sect193r1	c	r	-
96	$\mathbb{F}_{2^m}$	193	r	sect193r2	c	r	-
112	$\mathbb{F}_p$	224	k	secp224k1	c	r	-
112	$\mathbb{F}_p$	224	r	secp224r1	c	r	NIST P-224
112	$\mathbb{F}_{2^m}$	233	k	sect233k1	c	r	NIST K-233
112	$\mathbb{F}_{2^m}$	233	r	sect233r1	c	r	NIST B-233
115	$\mathbb{F}_{2^m}$	239	k	sect239k1	c	c	-
128	$\mathbb{F}_p$	256	k	secp256k1	c	r	-
128	$\mathbb{F}_p$	256	r	secp256r1	r	r	NIST P-256
128	$\mathbb{F}_{2^m}$	283	k	sect283k1	c	r	NIST K-283
128	$\mathbb{F}_{2^m}$	283	r	sect283r1	c	r	NIST B-283
192	$\mathbb{F}_p$	384	r	secp384r1	c	r	NIST P-384
192	$\mathbb{F}_{2^m}$	409	k	sect409k1	c	r	NIST K-409
192	$\mathbb{F}_{2^m}$	409	r	sect409r1	c	r	NIST B-409
256	$\mathbb{F}_p$	521	r	secp521r1	c	r	NIST P-521
256	$\mathbb{F}_{2^m}$	571	k	sect571k1	c	r	NIST K-571
256	$\mathbb{F}_{2^m}$	571	r	sect571r1	c	r	NIST B-571

Tabelle 2.2: *Sicherheitsniveau verschiedener ECC-Kurven [32]*

Zur Abschätzung der Performanz unterschiedlicher DSS-Verfahren, wurden in dieser Arbeit Performanz-Tests für verschiedene DSS-Verfahren durchgeführt.

Die nachfolgenden Performanzwerte (Tabellen 2.3 und 2.5) wurden mit einem *Raspberry Pi 3* Rechnersystem mit folgenden Informationen erstellt:

```
OS:                    Raspbian GNU/Linux 8 (jessie)
OS-Version:            Linux version 4.4.50-v7+
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Model name:            ARMv7 Processor rev 4 (v7l)
CPU max MHz:           1200,0000
CPU min MHz:           600,0000
```

Die Tabelle 2.3 zeigt das Ergebnis des Performanz-Tests für DSS-Verfahren. Gezeigt sind die erreichten Signierungen pro Sekunde und die erreichten Verifizierungen pro Sekunde, für verschiedene DSS-Verfahren mit unterschiedlichen Schlüssellängen (in Bit) und zusätzlich bei Verwendung von ECC, die SEC Bezeichnung der verwendeten ECC-Kurve. Die Werte wurden dabei mit OpenSSL ermittelt. Der verwendete Konsolen-Befehl, mit dem der Performanz-Tests ausgeführt wurde, lautet:

```
openssl speed rsa1024 dsa1024 rsa2048 dsa2048 rsa4096 ecdsa
```

Verfahren	Schlüssellänge	SEC Bezeichnung	Sign./s	Verifi./s
RSA	1024	-	254,3	5039,9
RSA	2048	-	40,0	1420,3
RSA	4096	-	5,8	372,8
DSA	1024	-	466,7	445,7
DSA	2048	-	134,7	123,4
ECDSA	160	secp160r1	1848,1	535,1
ECDSA	163	sect163k1	614,1	148,4
ECDSA	163	sect163r2	618,6	137,3
ECDSA	192	secp192r1	1493,3	402,4
ECDSA	224	secp224r1	1193,5	309,0
ECDSA	233	sect233k1	295,3	79,2
ECDSA	233	sect233r1	297,7	72,9
ECDSA	256	secp256r1	978,4	237,9
ECDSA	283	sect283k1	193,9	42,7
ECDSA	283	sect283r1	193,2	38,6
ECDSA	384	secp384r1	466,8	97,9
ECDSA	409	sect409k1	72,4	18,8
ECDSA	409	sect409r1	72,2	16,5
ECDSA	521	secp521r1	239,1	45,2
ECDSA	571	sect571k1	30,4	8,1
ECDSA	571	sect571r1	30,3	7,1

Tabelle 2.3: *Performanz einzelner DSS Verfahren*

Aus der Tabelle 2.3 ist ersichtlich, dass mit RSA zwar sehr schnell Signaturen verifiziert werden können, aber das die Erstellung einer Signatur verhältnismäßig sehr langsam ist. Da bei einer gegenseitigen Authentifizierung mit digitalen Signaturen von beiden Parteien sowohl eine Signatur erstellt wie validiert werden muss, ist die Performanz aus der Kombination aus beidem, der benötigte Vergleichswert. Bei einem Beispiel mit einem Sicherheitsniveau von 112 Bit lassen sich die Werte für RSA, DSA und ECDSA aus der Tabelle 2.3 gut miteinander vergleichen, siehe Tabelle 2.4. Dabei ist zu erkennen, das ECDSA die beste Performanz besitzt.

Verfahren	Schlüssellänge	SEC Bezeichnung	Sign./s	Verifi./s	(Sign. und Verifi.)/s
RSA	2048	-	40,0	1420,3	38,9
DSA	2048	-	134,7	123,4	64,4
ECDSA	224	secp224r1	1193,5	309,0	245,5

Tabelle 2.4: *Performanz-Vergleich RSA/DSA/ECDSA für kombinierte Signierung und Verifizierung*

Die Tabelle 2.5 zeigt das Ergebnis des Performanz-Tests für ECDH mit verschiedenen Kurven. Gezeigt sind die erreichten Operationen pro Sekunde (eine Operation entspricht einem Schlüsseltausch), für unterschiedliche ECC-Kurven. Die Werte wurden dabei wie-

der mit OpenSSL ermittelt. Der verwendete Konsolen-Befehl, mit dem der Performanz-Tests ausgeführt wurde, lautet:

```
openssl speed ecdh
```

Schlüssellänge (in Bit)	SEC Bezeichnung	Op./s
160	secp160r1	648,1
163	sect163k1	308,5
163	sect163r2	283,3
192	secp192r1	477,9
224	secp224r1	365,0
233	sect233k1	162,2
233	sect233r1	149,2
256	secp256r1	286,6
283	sect283k1	86,5
283	sect283r1	77,9
384	secp384r1	116,8
409	sect409k1	38,2
409	sect409r1	33,5
521	secp521r1	54,7
571	sect571k1	16,2
571	sect571r1	14,1

Tabelle 2.5: *Performanz von ECDH mit verschiedenen Kurven*

Aus den Daten der Tabellen 2.2, 2.3 und 2.5 geht hervor, dass die Nutzung von Primfeldern, nicht nur kleinere Schlüssellängen (bei gleichem Sicherheitsniveau) zur Folge hat (siehe Tabelle 2.2), sondern auch performanter bei der Erstellung und Validierung von digitalen Signaturen ist.

Bei einem geforderten Sicherheitsniveau von mindestens 128 Bit fällt dabei das Augenmerk auf die Kurve *secp256r1*, diese nutzt Primfelder und wird von allen neueren Standards empfohlen.

Die kleineren Schüssellängen und die auch daraus folgenden kleineren Signaturlängen ermöglichen es damit mehrere öffentliche Schüssel und auch Signaturen in einem einzigen EAP-Paket unterzubringen.

Und Ermöglichen es somit im Vergleich zu z.B EAP-TLS (wo zumeist RSA oder DSA zum Einsatz kommt) die Anzahl der benötigten EAP-Pakete für eine Authentifizierung massiv zu reduzieren, da keine vergleichsweise riesigen öffentlichen Schlüssel und Signaturen in Form von Zertifikaten übertragen werden müssen.

Dies spart aber nicht nur Bandbreite sonder auch Zeit, da wesentlich schneller eine Authentifizierung durchgeführt werden kann.

Ein weiterer großer Vorteil ergibt sich bei ECC und der Verwendung von DHE (ECDHE). Bei der ECDHE-Variante von ECDH, steht das E für den englischen Begriff *ephemeral* was im Deutschen so viel bedeutet wie *flüchtig*. Im Unterschied zum normalen (EC)DH, wird bei (EC)DHE für jeden Schlüsseltausch, neue Parameter (Schlüsselpaare) eingesetzt. Hier kommt ein großer Vorteil von ECC in Spiel, die Tatsache, dass Schlüsselpaare wesentlich schneller erzeugt werden können, als es beim normalen DH-Verfahren der Fall ist. Dies kommt daher das beim normalen DH zwei sehr große Primzufallszahlen erzeugt werden müssen, während bei ECDH eine normale große Zufallszahl ausreichend ist, um einen privaten Schlüssel zu erzeugen und im Anschluss daraus den öffentlichen Schlüssel zu berechnen.

Der ECDHE-Schlüsseltausch verfügt dabei über *forward secrecy* (im Deutschen etwa: vorwärtsorientierte Geheimhaltung), was so viel bedeutet wie Sicherheit die auch zu einem späteren Zeitpunkt besteht, wenn in der Zukunft, ein Langzeitschlüssel bekannt werden würde.

2.5.2 Symmetrisches Kryptosystem

2.5.2.1 Advanced Encryption Standard (AES)

Der Advanced Encryption Standard (AES) ist der aktuell praxisrelevanteste Blockchiffre. AES ist ein von der *National Institute of Standards and Technology* (NIST) standardisierter Kryptoalgorithmus (unter dem Namen *FIPS PUB 197* publiziert) [26] und wird aktuell noch als "das" symmetrische Verschlüsselungsverfahren der Wahl, für viele Anwendungsfälle (wie zum Beispiel TLS [11], SSH [19], IPsec [17], SIP [29], Kerberos [30] und SRTP [24]) eingesetzt. Für AES gibt es zu Zeit kein relevantes Angriffsverfahren und gilt als sehr sicheres Verschlüsselungsverfahren.

Des weitern ist AES nicht patentgeschützt und frei nutzbar, was mit ein Grund für die große Verbreitung des Algorithmus ist.

Für ein Sicherheitsniveau von 128 Bit ist eine (symmetrische) Schlüssellänge von 128 Bit erforderlich (AES128) [28].

2.5.2.2 Galois Counter Mode (GCM)

Der *Galois Counter Mode* (GCM)*Galois Counter Mode* ist ein Betriebsmodus für Blockchiffren (wie AES). Dabei ist GCM eine erweiterte Variante des Betriebsmodus *Counter Mode* (CTR), welcher mit einem GMAC über den Chiffrentext erweitert wurde, um neben der Vertraulichkeit (durch die Verschlüsselung) der Daten, auch die Authentizität und Integrität der Daten und zusätzlicher optionaler Authentifizierungsdaten sicherstellt. Damit entspricht GCM einem AEAD-Algorithmus (AEAD ist die englische Abkürzung für *Authenticated Encryption with Associated Data*).

Mehr Informationen zu AEAD finden sich in dem IETF-Standard *RFC 5116* [23].

Informationen zu GCM sind im IETF-Standard *RFC 5289* [31] und der NIST *Special Publication 800-38D* verfügbar.

Die Verwendung und die Größe des NonceExplicit (*nonce_explicit*) und des Salts sind in dieser Arbeit wie in *AES Galois Counter Mode (GCM) Cipher Suites for TLS* [33] beschrieben gewählt:

```
struct {
    opaque salt[4];
    opaque nonce_explicit[8];
} GCMNonce;
```

2.5.3 Key Derivation Function

Um aus dem mit ECDH ausgetauschtem Schlüssel, sichere Eingangsparameter für AES/GCM (wie symmetrische Schlüssel und Initialvektoren) zu erzeugen, bedarf es einer *Key Derivation Function* (KDF), die aus einem Eingangsschlüssel und gegebenenfalls weiteren Eingangsparametern, mehrere sichere und nicht zurückrechenbare Eingangsparameter erzeugt.

Pseudozufallszahlengeneratoren (im Englischen auch: pseudo random number generator; PRNG) sind dabei eine gute und oft verwendete Lösung für KDFs.

Dabei gilt, bei Pseudozufallszahlengeneratoren, dieselbe Annahme wie bei allen anderen Kryptoverfahren, nur öffentlich und langfristig bekannte Verfahren, mit denen sich schon Hunderte von Kryptologen beschäftigt haben, können als wirklich sicher angesehen werden.

Somit fällt hier das Hautaugenmerk auf die *Pseudorandom Function* (PRF; im Deutschen: Pseudozufalls-Funktion) aus dem TLS-Standard. Dieser Pseudozufallszahlengeneratoren ist, durch den Einsatz von TLS in HTTPS, auf unzähligen Milliarden Geräten im Einsatz und kann somit (mit dem Hintergrund, dass es keine bekannte Angriffsstrategie gibt) als sehr sicher angesehen werden.

2.5.3.1 Die TLS Pseudorandom Function (PRF)

Die TLS Pseudorandom Function (PRF) des TLS-Standards wird im TLS-Protokoll eingesetzt um aus einem geteilten Geheimnis (im Englischen: shared secret) zwischen Client und Server, einen *master secret* und aus diesem wiederum die Cipher-Parameter zu erzeugen. Dazu nutzt die PRF den HMAC-Algorithmus.

Die im HMAC-Algorithmus verwendete Hashfunktion ist seit TLS Version 1.2 in der *Cipher Suite* angegeben und nicht mehr statisch festgelegt. Lauf TLS Spezifikation ist der Hash-Algorithmus SHA-256 die passende Wahl bei der Verwendung von AES-128 und GCM [31].

Das Sicherheitsniveau bei der Verwendung von SHA-256 (in KDFs), entspricht dabei laut NIST, mindestens 256 Bit [28] [21]. Und erfüllt somit bei Weitem das Sicherheitsniveau dieser Arbeit.

Für Informationen zur Funktionsweise und dem grundsätzlichen Aufbau der PRF siehe *RFC 5246* [13].

2.5.4 Cipher Suite

Eine Cipher Suite (zu Deutsch: Chiffrensammlung) ist in TLS eine Zusammenstellung von kryptografischen Verfahren, die zwischen Client und Server ausgehandelt werden, um diese zum Aufbau einer verschlüsselten Verbindung einzusetzen.

Die in dieser Voruntersuchung für diese Arbeit festgelegten Kryptosysteme entsprechen dabei der TLS 1.2 Cipher Suite `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` [31]. Diese gilt als sehr sicher, was zeigt, dass die grundsätzliche Kombination dieser kryptografischen Verfahren keine Sicherheitsrisiken mit sich bringt.

Kapitel 3

Konzept

Dieses Kapitel behandelt das Konzept des Blockchain-basierten Triple-A-Systems.

Um eine Authentifizierung zu einer Netzwerknutzung (wie WLAN) mit einem Blockchain-basierten AAA-System durchzuführen, ist das *Extensible Authentication Protocol* (EAP) ein Mittel der Wahl siehe dazu Kapitel 1.4. Dazu wird eine eigene EAP-Methode benötigt, die zum einen ein geeignetes Authentifizierungsverfahren bzw. Authentifizierungsprotokoll implementiert, dass zur Authentifizierung mit einem Blockchain-basierten AAA-System geeignet ist.

Zum anderen muss die EAP-Methode in der Lage sein, direkt mit dem Blockchain-Netzwerk zu interagieren oder einen Blockchain-Clienten zu nutzen, um dies zu tun.

Hier fällte die Wahl auf Zweites, da die Einbindung des Ethereum-Clienten (mit der Bezeichnung *geth*) einen geringeren Aufwand bedeutet als eine Integration eines Ethereum-Clienten in die EAP-Methode und zum anderen der Aktualisierungsaufwand der EAP-Methode massiv sinkt, da diese nicht bei jeder Aktualisierung des Blockchain-Clienten auch aktualisiert werden muss.

Oder anders gesagt, dass der Aktualisierungszyklus der EAP-Methode nicht von dem Aktualisierungszyklus des Blockchain-Clienten abhängig ist.

Die EAP-Methode ermöglicht es somit, den Zugriff auf ein Netzwerk (in Falle dieser Arbeit ein WLAN-Netzwerk) zu regeln. Der IEEE 802.1X Standard bildet dabei die Schnittstelle zwischen EAP und dem der darunterliegenden Netzwerkebene (bei WLAN durch den IEEE 802.11 Standard geregelt).

Folgende Parteien sind involviert:

Supplicant
: Nutzer, der mit einem Geräte den Zugriff auf ein Netzwerk erbittet. Im 802.1X Standart wird die Bezeichnung *Supplicant*, für die Instanz die sich mit einem Netzwerk verbinden möchte, verwendet. In im EAP-Standart und nachfolgend in diesem Dokument als *Peer* bezeichnet.

Authenticator
: Komponente, die im Netzwerk-Zugangspunkt die Authentifikation übernimmt. Im EAP-Protokoll, die Instanz die die Authentifikation initialisiert. Im Fall von 802.11 (WLAN) der (WLAN-)Access-Point.

Authentication Server

Smart-Contract (der Blockchain) der die Aufgabe eines sogenannten *Authentication Server* (aus dem Backend des AAA-Services) übernimmt. Wobei hier angemerkt sei, dass es sich dabei um keinen *Authentication Server* aus dem EAP-Standard handelt, da dieser nicht direkt das EAP-Protokoll verwendet.

CA

Die Zertifizierungsstelle (im Englischen: Certificate Authority; CA) ist im Kontext dieser Arbeit (EAP-BCA-Protokoll) die Institution, die für die Authenticators die Schlüssel-Autorisierungs-Signaturen ausstellt und somit auch die einzelnen Authenticator autorisiert.

Abbildung 3.1 zeigt eine Übersicht der Protokollschichten des WLAN-Clients und des Access-Points bei Verwendung des Blockchain-basierten Triple-A-Systems.

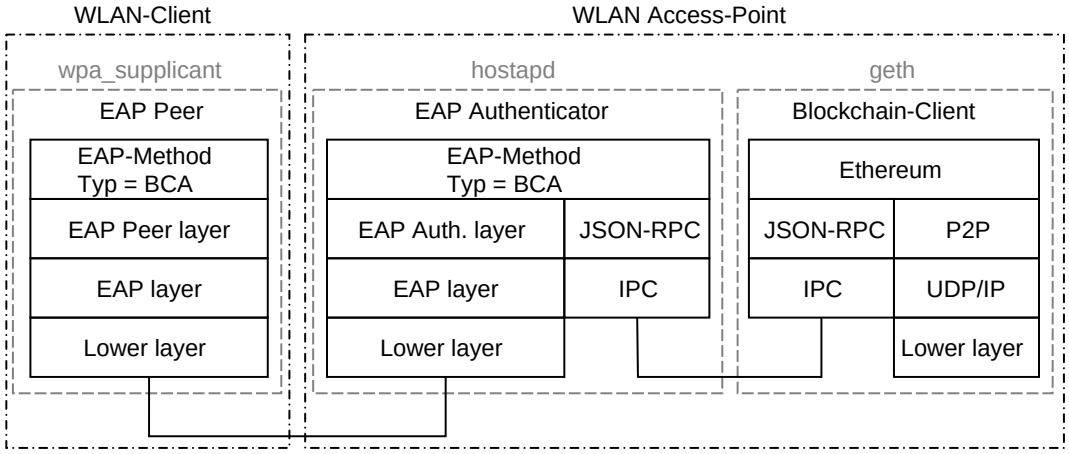


Abbildung 3.1: *Übersicht der Protokollschichten des Blockchain-basierten Triple-A-System*

3.1 EAP-Blockchain-Authorisation (EAP-BCA) Protokoll

3.1.1 Terminologie

In diesem Abschnitt werden, neben den schon erwähnten Bezeichnungen der involvierten Parteien, häufig folgende Begriffe verwendet:

Peer

Instanz die sich für einen Netzwerkzugriff authentifizieren möchte und dem Authenticator Antwortet. Im 802.1X Standart wird der Peer als Supplicant (im Deutschen: Bittsteller oder Antragsteller) bezeichnet.

AAA

Abgekürzt für Authentifizierung, Autorisierung und Abrechnung (im Englischen: Authentication, Authorization and Accounting). Siehe 1.3 *Triple-A-Systeme*.

Master Session Key (MSK)
Schlüsselmaterial (mit 64 Bytes Länge) das zwischen dem EAP Peer und Server ausgehandelt wird und von der EAP-Methode zurückgegeben wird. In bestehenden Implementierungen agiert der AAA-Server als EAP-Server und sendet den MSK an den Authenticator [8]. Dieser kann den MSK zur Sicherung der Verbindung zwischen dem Peer und dem vom Authenticator genehmigten Netzwerk einsetzen.

Extended Master Session Key (EMSK)
Zusätzliches Schlüsselmaterial (mit 64 Bytes Länge) das zwischen dem EAP Peer und Server ausgehandelt wird und von der EAP-Methode zurückgegeben wird. Der EMSK wird nicht an den Authenticator oder eine dritte Partei bekannt gegeben [8]. Laut EAP-Standart ist der EMSK für den späteren Einsatz Reserviert und zur Zeit nicht näher Definiert [8].

3.1.2 Zeitstempel und Zufallszahlen

Als Zeitstempel werden in dieser Arbeit moderne 64-Bit Unix-Zeitstempel verwenden, genauer gesagt eine 8 Byte Lange natürliche Zahl ohne Vorzeichen (`uint64`) die die Anzahl der Millisekunden, die seit dem 1.1.1970 um 00:00 Uhr (UTC) vergangen sind angibt.

Anstatt direkte erzeugte Zufallszahlen werden in dieser Arbeit, die Hashwerte der öffentlichen ECDH(E)-Schlüssel verwendet. Diese Hashwerte erfüllen die gleichen Bedingungen wie jede andere direkt erzeugte sichere Zufallszahl, da der Quell-Konsens auf einer kryptografisch-sicheren Zufallszahl (privater ECDH(E)-Schlüssel) basiert, der bei jeder Verbindung variiert.

Die "Zufallszahlen" (Hashwerte der öffentlichen ECDH(E)-Schlüssel) dienen der Absicherung gegen Replay-Angriffe (im Deutschen: Wiederhohlungs-Angriffe) bzw. der Replay-Protection also dem Schutz gegen solche Angriffe. Bei Replay-Angriffen werden die gleichen EAP-Pakete von einem Angreifer erneut gesendet, um sich gegenüber einem Authenticator oder einem Peer zu Authentifizieren.

Die Kombination aus Zeitstempel und Zufallszahl oder das integrieren eines Zeitstempels in eine Zufallszahl sind übliche Vorgehensweisen um die Einmaligkeit einer Zufallszahl sicherzustellen und unter gewissen Bedingungen auch vom Gegenüber überprüfbar zu machen. Eine dieser Bedingungen kann sein das die Zufallszahl nur einen gewissen Gültigkeitszeitraum besitzt.

Ein Beispiel dazu sind die Zufallszahlen, die bei TLS zwischen Client und Server ausgetauscht werden (siehe *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, Seite 39, [13]). Diese 32 Byte Langen Zufallszahlen bestehen eigentlich nur auch einer Gruppe von 28 zufälligen Bytes, denen ein 4 Byte langer (Standart)

Unix-Zeitstempel vorausgeht.

In dieser Arbeit wird ein Zeitstempel (`timestamp`) immer in Kombination mit einem Hashwert des öffentlichen ECDH(E)-Schlüssels (Zufallszahl) verwendet. Somit lassen sich beide Werte auch als Einheit (wie im TLS-Standart) betrachten.

Der Zeitstempel für sich, entspricht dadurch nicht nur dem Zeitpunkt des Authentifizierungs-Starts, sondern auch dem Zentrum eines Zeitfensters in dem die ausgetauschten Authentifizierungs-Signaturen und die damit gesichtern Parameter gültig sind. Da sich darunter auch die Zufallszahlen befinden, kann davon ausgegangen werden das innerhalb des Gültigkeitszeitraums die Kombination aus Zeitstempel und Zufallszahl nur ein einzelnes mal vorkommt. Die Authentifizierungs-Signaturen sind somit auch Unikarte die nicht wiederverwendet werden können um Replay-Angriffe durchzuführen.

3.1.3 EAP-Methode: Identity

Typischerweise wird vor einer EAP-Authentifizierungs-Methode, ein Request und Response vom Typ *Identity*, zwischen Authenticator und Peer ausgetauscht. Dies kommt daher, dass viele EAP-Authentifizierungs-Methode, Zugriff auf die Identität des Nutzers (Nutzer-ID; im Englischen: user-Id) möchten bzw. benötigen, welche sie durch diesen Austausch erhalten.

Da bei der EAP-Methode *Identity* die Nutzer-ID unverschlüsselt übertragen wird, wird bei allen neueren EAP-Methoden (wie EAP-TLS, PEAP und beim Tunneln über EAP-TTLS) eine öffentliche Nutzer-ID verwendet, um die eigentliche Nutzer-Id vor der Ausspähung zu schützen (Identitätsschutz; im Englischen: Identity Protection).

Um diesen Paketaustausch nicht zu verschwenden, kann in EAP-BCA diese öffentliche Nutzer-ID dazu verwendet werden, dem Authenticator im Vorfeld mitzuteilen, welche bcNetworkId für die Verbindung verwendet werden soll.

Diese Information ist kein Identitätsmerkmal des Peers bzw. Nutzers, solange keine bcNetworkId einem bestimmten Nutzer zuzuordnen ist, also solange die Anzahl der Nutzer pro bcNetworkId größer als 1 ist.

Anhand der bcNetworkId ist der Authenticator in der Lage, eine unterschiedliche Konfigurationen zu verwenden, die am Ende Auswirkungen auf z.B. das Subnetz für den Peer oder den für die Autorisierung verwendeten Smart Contract, haben.

Somit lassen sich dann z.B. bei einer Verwendung in einem WLAN, trotz der gleichen SSID, unterschiedliche Sub-Netze, Nutzer-Realms, usw. zuordnen.

3.1.4 Daten Präsentationssprache (Pseudocode)

Als Präsentationssprache bzw. Pseudocode das nachfolgende Protokolls, wird in diesem Dokument, die Syntax der Programmiersprache Go verwendet, die minimal Erweitert wurde. Dies soll für eine bessere Lesbarkeit sorgen und somit ein schnelleres Nachvoll-

ziehen der angegeben Daten- bzw. Protokollstrukturen ermöglichen.

Die Ergänzungen sind nachfolgend beschrieben.

3.1.4.1 Konkatenation (Verkettung)

Um den Pseudocode möglichst Kompakt und Übersichtlich zu halten werden Konkatenationen (Verkettungen; von z.B. Strings und Arrays) mit dem Symbol ∘ dargestellt.

3.1.4.2 Grundsätzliche (De-/)Serialisierung der Daten

Bei der Serialisierung bzw. Deserialisierung der Daten gilt die für Netzwerkprotokolle übliche Byte-Reihenfolge *big-endian*.

Der Datentype *bool* wird als `uint8` mit den Werten 1 für `true` und 0 für `false` während der Serialisierung dargestellt.

3.1.4.3 Arrays

Der größte Teil der für das Protokoll festgelegten, auf Arrays basierenden Datentypen besitzt eine festgelegte Länge. Und die Strukturen der einzelnen Protokoll-Nachrichten sehen jeweils, maximal ein Array mit variabler Länge vor.

Dadurch enthält jede serialisierte Nachricht des Protokolls, maximal ein variables Array. Bei der Deserialisierung der Nachrichten ist es somit immer möglich, die Länge dieses einen variablen Arrays anhand der EAP-Paketlänge abzüglich der fixen Länge der Nachrichtenstruktur des jeweiligen (in der Nachricht mit angegebenen) Nachrichtentypes, zu berechnen.

3.1.4.4 Strukturen

Um Variationen von Strukturen im Pseudocode abbilden zu können sind mehrere syntaktische Erweiterungen hinzugekommen.

Darunter sind ...

... Bedingungen für Existenz von Elementen, Beispiel:

```
type IfStructureElement struct {
    elementA        uint32
    hasElementB     bool
    if (.hasElementB) {
        elementB    int8
    }
}
```

Das Element `elementB` ist nur in der Struktur (und in Instanzen) enthalten, wenn `hasElementB true` ist.

... Fall-Unterscheidung für den Datentyp eines Elements, Beispiel:

```
type CaseStructureElementType struct {
    elementA        int32
    elementBType    uint8
    elementB        select (.elementBType) {
        case 0: int8
        case 1: int16
        case 2: int32
    }
}
```

Der Datentyp des Elements `elementB` ist abhängig von dem Wert des Elements `elementBType`.
Beispiel: Wenn der Wert von `elementBType = 2` ist, ist der Datentyp des Elements `elementB int32`.

... Fall-Unterscheidung für den Datentyp eines Elements, bei der Initialisierung einer Struktur Variable, Beispiel:

```
var i8      int8
var i16     int16
var i32     int32

// ...

varType := 1
structure := CaseStructureElementType{
    1000,
    varType,
    select (varType) {
        case 0:     i8
        case 1:     i16
        case 2:     i32
    }
}
```

Je nach Wert des Elements `varType` wird das Element `elementB` mit der Variable `i8`, `i16` oder `i32` initialisiert. Im Fall des gezeigten Beispieles wird `elementB` mit `i16` initialisiert.

3.1.5 Protokoll Konstanten

Für das Protokoll sind einige Konstanten festgelegt:

```
ECDHKeyBitLength    := 256
ECDHKeyLength       := ECDHKeyBitLength / 8
ECDSAKeyBitLength   := 256
ECDSAKeyLength      := ECDSAKeyBitLength / 8
KeyHashLength       := 32
CipherNonceExLength := 8
CipherKeyBitLength  := 128
CipherKeyLength     := CipherKeyBitLength / 8
CipherSaltLength    := 4
KeyBlockLength      := CipherKeyLength + CipherSaltLength
```

```
MasterSecretLength  := 48

MSKLength           := 64
EMSKLength          := 64
MSKBlockLength      := MSKLength + EMSKLength

TimestampMaxDelta   := 300
```

Beschreibung:

ECDHKeyBitLength
Schlüssel-Länge (in Bits) für die, im ECDH-Verfahren eingesetzten privaten Schlüssel.

ECDHKeyLength
Schlüssel-Länge (in Bytes) für die, im ECDH-Verfahren eingesetzten privaten Schlüssel.

ECDSAKeyBitLength
Schlüssel-Länge (in Bits) für die, im ECDSA-Verfahren eingesetzten privaten Schlüssel.

ECDSAKeyLength
Schlüssel-Länge (in Bytes) für die, im ECDSA-Verfahren eingesetzten privaten Schlüssel.

KeyHashLength
Länge (in Bytes) eines Hashwerts eines öffentlichen EC-Schlüssels.

CipherNonceExLength
Länge (in Bytes) des im *AEADCipher* verwendeten *nonceExplicit*.

CipherKeyBitLength
Länge (in Bits) des *AEADCipher*-Schlüssels.

CipherKeyLength
Länge (in Bytes) des *AEADCipher*-Schlüssels.

CipherSaltLength
Länge (in Bytes) des im *AEADCipher* verwendeten *Salts*.

KeyBlockLength
Länge (in Bytes) des mit der PRF erzeugten `keyBlock`.

MasterSecretLength
Länge (in Bytes) des MasterSecret.

MSKLength
Länge (in Bytes) des *Master Session Key* (MSK) des EAP-Methoden Schlüsselmaterials.

EMSKLength
Länge (in Bytes) des *Extended Master Session Key* (EMSK) des EAP-Methoden Schlüsselmaterials.

MSKBlockLength
Länge (in Bytes) des mit der PRF erzeugten `mskBlock`.

TimestampMaxDelta
Maximal erlaubte Zeitdifferenz bzw. Zeitraum die der Zeitstempel (`timestamp`) von der aktuellen Gerätezeit abweichen darf.

3.1.6 Datentypen Deklaration

```
type Timestamp              uint64
type ECDHInteger            [ECDHKeyLength]byte
type ECDSAInteger           [ECDSAKeyLength]byte
type KeyHash                [KeyHashLength]byte
type ECDHPrivateKey         ECDHInteger
type ECDSAPrivateKey        ECDSAInteger

type ECDHPoint struct {
    x   ECDHInteger
    y   ECDHInteger
}

type ECDSAPoint struct {
    x   ECDSAInteger
    y   ECDSAInteger
}

type ECDHPublicKey  ECDHPoint
type ECDSAPublicKey ECDSAPoint

type ECDSASignature struct {
    r   ECDSAInteger
    s   ECDSAInteger
}

type AEADCipher struct {
    nonceExplicit   [CipherNonceExLength]byte
    aeadCiphered    []byte
}

type CipherParameters struct {
    key     [CipherKeyLength]byte
    salt    [CipherSaltLength]byte
}
```

3.1.7 Funktionsprototypen

3.1.7.1 Funktionsprototypen des Authenticator & Peer

```
func timeNow() Timestamp
func error() void

func ecdhCreatePrivateKey() ECDHPrivateKey
func ecdhCreatePublicKey(privateKey ECDHPrivateKey) ECDHPublicKey
func hashDHKey(key ECDHPublicKey) KeyHash

func ecdsaCreatePrivateKey() ECDSAPrivateKey
func ecdsaCreatePublicKey(privateKey ECDSAPrivateKey) ECDSAPublicKey
func hashDSAKey(key ECDSAPublicKey) KeyHash

func ecdhGenerateKey(privateKey ECDHPrivateKey, publicKey ECDHPublicKey) []byte
```

```
func ecdhGenerateKeyByDSAKeys(privKey ECDSAPrivateKey, pubKey ECDSAPublicKey) []byte

func ecdsaSign(msg []byte, privKey ECDSAPrivateKey) ECDSASignature
func ecdsaValidate(msg []byte, sign ECDSASignature, pubKey ECDSAPublicKey) bool

func prf(secret []byte, label string, seed []byte, length uint64) []byte

func cipherEncryption(msg []byte, cipherParameters CipherParameters) AEADCipher
func cipherDecryption(cipher AEADCipher, cipherParameters CipherParameters) []byte
```

Beschreibung:

`timeNow() Timestamp`
: Gibt einen Zeitstempel (im Englischen: Timestamp) zurück, der die Anzahl der Millisekunden die seit dem 1.1.1970 um 00:00 Uhr (UTC) vergangen sind, angibt.

`error()`
: Der Aufruf beendet die EAP-Authentifizierung mit einem Fehler.

`ecdhCreatePrivateKey() ECDHPrivateKey`
: Erzeugt und gibt einen Privaten-Schlüsse zurück, dabei werden die ECDH-Kurven-Parameter verwendet.

`ecdhCreatePublicKey(privateKey ECDHPrivateKey) ECDHPublicKey`
: Erzeugt und gibt einen öffentlichen Schlüssel zurück. Dieser wird Anhand des Privaten-Schlüsses `privateKey` erzeugt, wobei die ECDH-Kurven-Parametern verwendet werden.

`hashDHKey(key ECDHPublicKey) KeyHash`
: Erzeugt den Hash des öffentlichen ECDH-Schlüssels `key` und gibt diesen zurück.

`ecdsaCreatePrivateKey() ECDSAPrivateKey`
: Erzeugt und gibt einen Privaten-Schlüsse zurück, dabei werden die ECDSA-Kurven-Parameter verwendet.

`ecdsaCreatePublicKey(privateKey ECDSAPrivateKey) ECDSAPublicKey`
: Erzeugt und gibt einen öffentlichen Schlüssel zurück. Dieser wird Anhand des Privaten-Schlüsses `privateKey` erzeugt, wobei die ECDSA-Kurven-Parametern verwendet werden.

`hashDSAKey(key ECDSAPublicKey) KeyHash`
: Erzeugt den Hash des öffentlichen ECDSA-Schlüssels `key` und gibt diesen zurück.

`ecdhGenerateKey(priv ECDHPrivateKey, pub ECDHPublicKey) []byte`
: Gibt einen mit ECDH erzeugten Schlüssel zurück. Für ECDH werden die Schlüssel `privateKey` und `publicKey` mit den ECDH-Kurven-Parametern verwendet.

`ecdhGenerateKeyByDSAKeys(...) []byte`
: Gibt einen mit ECDH erzeugten Schlüssel zurück. Für ECDH werden die Schlüssel `privateKey` und `publicKey` mit den ECDSA-Kurven-Parametern verwendet.

`ecdsaSign(msg []byte, privKey ECDSAPrivateKey) ECDSASignature`
: Erzeugt eine ECDSA-Signatur über die Nachricht `msg` mit dem Privaten-Schlüssel `privKey` und gibt diese zurück.

`ecdsaValidate(...) bool`
: Überprüft die ECDSA-Signatur `sign` mit der signierten Nachricht `msg` und dem öffentlichen Schlüssel des Signierers `pubKey` und gibt `true` zurück sollte die Signatur Korrekt sein, ansonsten `false`.

`prf(secret []byte, label string, seed []byte, length uint64) []byte`
: Erzeugt mit einem Pseudo-Zufallszahlen-Generator, der die Parameter `secret` (geheimer Schlüssel), `label` und `seed` als Eingangs-Parameter erhält, eine Pseudo-Zufallszahl als Byte-Array der Länge `length` und gibt diese zurück.

`cipherEncryption(...) AEADCipher`
: Verschlüsselt die Nachricht (Kartext) `msg` mit den Cipher-Parametern `cipherParameters` und gibt den Chiffretext als `AEADCipher`-Struktur-Instanz zurück.

`cipherDecryption(...) []byte`
: Entschlüsselt den Chiffretext `cipher` mit den Cipher-Parametern `cipherParameters` und gibt den Klartext als Byte-Array zurück.

3.1.8 Kryptografische Funktionen

Die hier eingesetzten kryptografischen Funktionen sollen die Sicherheit des Protokolls gewährleisten. Nachfolgend sind die Sicherheitsaspekte der einzelnen Funktionen aufgezeigt.

3.1.8.1 `hashDSAKey`-Methode

Die Methode `hashDSAKey` erzeugt den Hashwert eines öffentlichen ECDSA-Schlüssels. Der dazu verwendete Hash-Algorithmus ist *SHA-256*. Das Sicherheitsniveau eines *SHA-256*-Hashwerts entspricht dabei mindestens 128 Bit [28].

Über die Hashwerte wird sichergestellt, dass der Peer vom Authenticator den korrekten (nicht Kompromittierten) öffentlichen Schlüssel der CA (`caPublicKey`) erhält. Da der Hashwert des übertragenen öffentlichen Schlüssels mit dem in der Konfiguration des Peers hinterlegtem Hashwert der CA (`caPublicKey`) verglichen und somit überprüft werden kann.

3.1.8.2 `prf`-Methode

Die Methode `prf` erzeugt aus dem mit ECDH(E) ausgetauschtem Schlüssel die Parameter für den AEADCipher, siehe dazu 2.5.3.1 *Die TLS Pseudorandom Function (PRF)*.

3.1.8.3 `cipherEncryption`- und `cipherDecryption`-Methode

Die Methode `cipherDecryption` entschlüsselt nicht nur die mit der Methode `cipherEncryption` verschlüsselten Daten sondern stellt auch die Integrität der Daten sicher, siehe dazu 2.5.2.2 *Galois Counter Mode (GCM).*

3.1.8.4 `ecdh*`- und `ecdsa*`-Methode

Die grundlegenden Sicherheitsaspekte der ECDH- und ECDSA-Funktionen sind im Kapitel 2 *Voruntersuchung* unter 2.5.1.2 *Elliptische-Kurven-Kryptografie (ECC)* zu finden.

Während (die Methoden des) ECDH(E) für den Austausch eines sicheren symmetrischen Schlüssels sorgt und somit zur Erstellung eines symmetrisch-verschlüsselten Kanals. Sichert (die Methoden des) ECDSA die Authentizität der Endpunkte (Authenticator und Peer) ab und verhindert somit einen Man-in-the-Middle-Angriff.

3.1.8.5 `ecdsaValidate`-Methode

Alle im Protokoll verwendeten Signaturen werden mit der Methode `ecdsaValidate` überprüft. Sollte eine Signatur nicht korrekt sein, so wirft die Methode ein `false` als Return-Wert. Dieser Return-Wert wird im nachfolgenden Programmcode mit einer Bedingten Programm Anweisung überprüft und im Falle von `false` wird die Authentifizierung abgebrochen.

Die verwendeten und überprüften Signaturen sind:

Die **Authenticator-Authentifikation-Signatur** (`authAuthenticationSign`) bestätigt gegenüber dem Peer die Identität des Authenticators, da der Authenticator mit einer korrekten Signierung des gemeinsamen Zeitstempels (`timestamp`) und des öffentlichen ECDH-Schlüssels des Authenticators (`ecdhAuthPublicKey`) bestätigt, dass er im Besitz des privaten Schlüssels des Authenticators (`authPrivateKey`) ist. Gleichzeitig wird, die Integrität des öffentlichen ECDH-Schlüssels des Authenticators (`ecdhAuthPublicKey`) sichergestellt. Somit ist es für den Peer möglich einen Man-in-the-Middle-Angriff auszuschließen, da die Signatur ungültig ist falls der `sharedKey` (der vom `ecdhAuthPublicKey` abhängig ist) nicht mit dem über ECDH(E) ausgehandelten und verwendeten übereinstimmt.

Mit der, von der CA für den öffentlichen Schlüssel des Authenticators ausgestellten, **Schlüssel-Autorisierungs-Signatur** (`authPublicKeySign`) ist es möglich die Integrität (des öffentlichen Schlüssels des Authenticators) und die Autorisierung des Authenticators zu überprüfen.

Die **Peer-Authentifikation-Signatur** (`peerAuthenticationSign`) bestätigt gegenüber dem Authenticator die Identität des Peer, da der Peer mit einer korrekten Signierung des gemeinsamen Zeitstempels (`timestamp`) und des mit ECDH ausgetauschten Schlüssels (`sharedKey`) bestätigt, dass er im Besitz des privaten Schlüssels des Peers (`peerPrivateKey`) ist. Gleichzeitig wird, die Integrität des mit ECDH ausgetauschten Schlüssels (`sharedKey`) sicherstellt. Somit ist es für den Authenticator möglich einen Man-in-the-Middle-Angriff auszuschließen, da die Signatur ungültig ist falls der `sharedKey` nicht mit dem über ECDH(E) ausgehandelten und verwendeten übereinstimmt.

3.1.9 Eingangsparameter des Authenticators

3.1.9.1 Parameter aus den Konfigurationsdaten

```
var bcNetworkId         []byte
var authPrivateKey      ECDSAPrivateKey
```

3.1.9.2 Initialisierung

Einige Parameter werden bei der Initialisierung des Authenticators erzeugt und müssen somit nicht in den Konfigurationsdaten des Authenticators enthalten sein bzw. für dessen Konfiguration angegeben werden.

Pseudocode zur Initialisierung des Authenticators:

```
authPublicKey := ecdsaCreatePublicKey(authPrivateKey)
```

Andere Parameter sind durch den Smart Contract abrufbar, darunter sind:

```
var caPublicKey         ECDSAPublicKey
var authPublicKeySign   ECDSASignature
```

3.1.10 Eingangsparameter des Peer

3.1.10.1 Parameter aus den Konfigurationsdaten

```
var bcNetworkId         []byte
var caKeyHash           KeyHash
var peerPrivateKey      ECDSAPrivateKey
```

3.1.10.2 Initialisierung

Einige Parameter werden bei der Initialisierung des Peer erzeugt und müssen somit nicht in den Konfigurationsdaten des Peer enthalten sein bzw. für dessen Konfiguration angegeben werden.

Pseudocode zur Initialisierung des Peer:

```
peerPublicKey := ecdsaCreatePublicKey(peerPrivateKey)
peerKeyHash   := hashDSAKey(peerPublicKey)
```

3.1.11 Protokoll Ablauf

Der Ablauf der EAP-BCA-Methode bzw. des EAP-BCA-Protokolls ist als Übersicht, mit den jeweiligen Schritten, in der Abbildung 3.2 gezeigt.

Das EAP-BCA-Protokoll besteht jeweils nur aus einem EAP-Request und einem EAP-Response.

Der Ablauf des EAP-Protokolls ist bei Nutzung der BCA-Methode der Folgende:

Der **1. und 2. Schritt** im EAP-Protokoll ist der Austausch eines EAP-Request und EAP-Response vom Typ *Identity*. Siehe dazu 3.1.3 *EAP-Methode: Identity*.

Im **3. Schritt** des EAP-Protokolls startet die EAP-BCA-Methode und somit wird die erste Nachricht (EAP-Request) des BCA-Protokolls, die Inital-Nachricht (InitMsg) vom Authenticator an den Peer versendet. Siehe dazu 3.1.11.1 *Inital-Nachricht (InitMsg)*.

Nachdem der Peer die InitMsg erhalten hat, ist dieser in der Lage mit seinem neu erstellten privaten ECDH(E)-Schlüssel (`ecdhPeerPrivateKey`) und dem öffentlichen ECDH(E)-Schlüssel das Authenticators (`ecdhAuthPublicKey`) einen ECDH-Schlüsseltausch vorzunehmen und somit den `sharedKey` zu errechnen.

Auch erzeugt der Peer, mit Hilfe des Diffie-Hellman-Protokolls (wobei die ECDSA-Kurvenparameter verwendet werden), des öffentlichen ECDSA-Schlüssels des Authenticators (`authPublicKey`) und eines neu erzeugten EC-Schlüsselpaares (`ecEncPrivateKey` und `ecEncPublicKey`), einen weiteren Schlüssel (`ecEncSharedKey`). Dieser Schlüssel ermöglicht eine Verschlüsselung die nur vom Inhaber der privaten ECDSA-Schlüssels des Authenticators (`authPrivateKey`) entschlüsselt werden kann.

Mit dem `sharedKey`, dem `ecEncSharedKey`, dem Hashwert seines öffentlichen ECDH(E)-Schlüssels (`ecdhPeerKeyHash`) und dem Hashwert des öffentlichen ECDH(E)-Schlüssels des Authenticators (`ecdhAuthKeyHash`), lassen sich nun die Cipher-Parameter `cipherParameters` erzeugen, die nachfolgend zur Verschlüsselung benötigt werden.

Als Antwort auf die InitMsg (den EAP-Request) des Authenticators sendet der Peer in **Schritt 4** einen EAP-Response mit einer Authentifizierungs-Nachricht (AuthMsg) an den Authenticator. Siehe dazu 3.1.11.2 *Authentifizierungs-Nachricht (AuthMsg)*.

Nachdem erhalten der AuthMsg ist auch der Authenticator in der Lage, den `sharedKey`, den `ecEncSharedKey` und am Ende die Cipher-Parameter (`cipherParameters`) zu

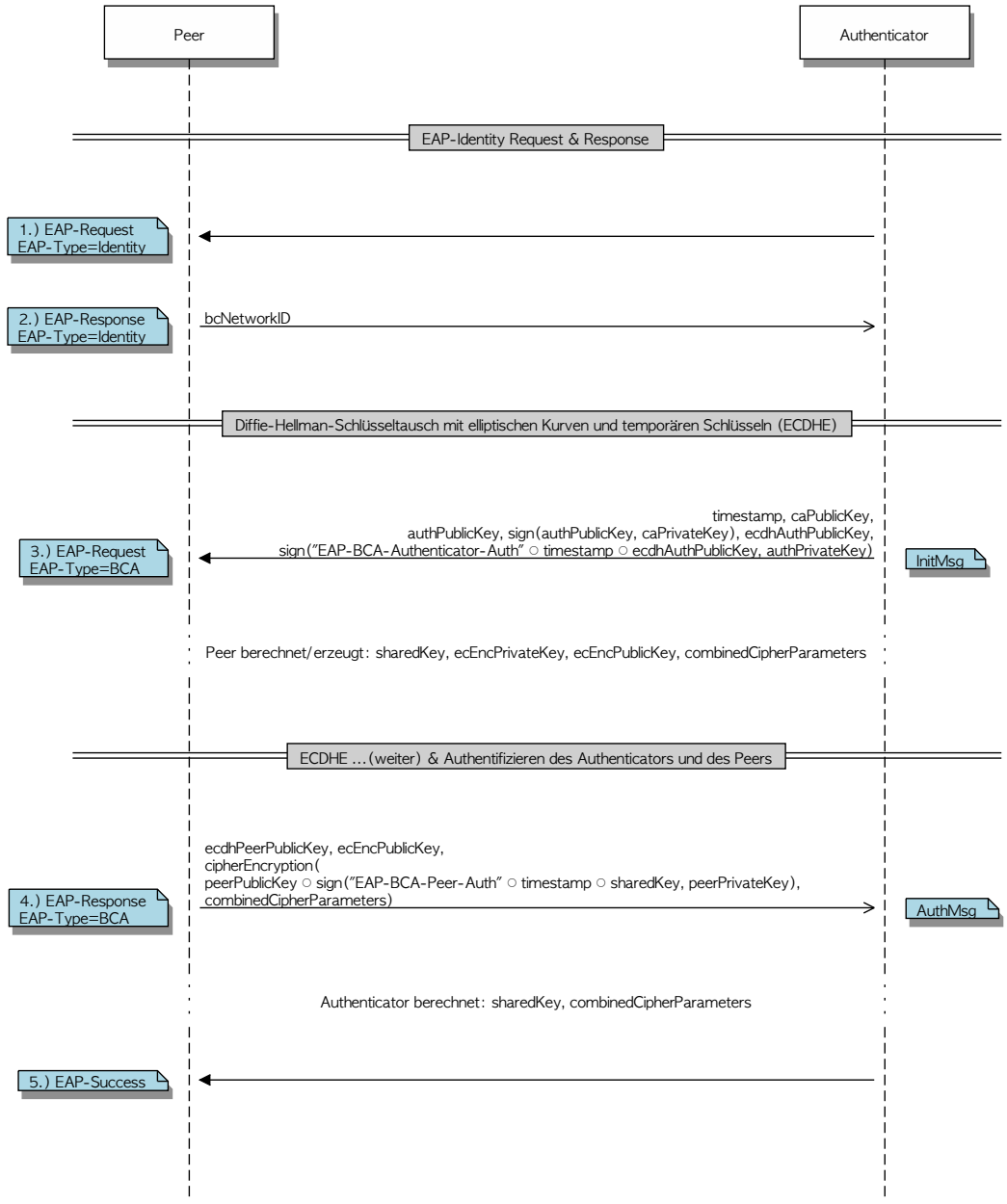


Abbildung 3.2: *EAP-Authentifizierung mit der EAP-BCA-Methode (EAP-BCA Protokoll)*

erzeugen. Um Schliesslich, die mit den `cipherParameters`, den verschlüsselten Nachrichtenteil zu entschlüsseln.

Der, im verschlüsselten Nachrichtenteil, enthaltene öffentliche ECDSA-Schlüssel des Peer (`peerPublicKey`) in verbindung mit der Peer-Authentifikation-Signatur (`peerAuthenticationSign`) ermöglicht eine Authentifizierung des Peer.

Der Hashwert des öffentlichen Schlüssels des Peer (`peerKeyHash`) dient als Identifikationsmerkmal und wird verwendet um mit Hilfe des *AAA-Contracts* die Autorisierung des Peers für den Netzwerkzugriff abzufragen (siehe 3.2.1 *AAA-Contract*).

Im **5. Schritt** und letzten Schritt der EAP-Authentifizierung wird bei erfolgreicher Authentifizierung mit der EAP-BCA-Methode (bzw. erfolgreicher Autorisierung über den AAA-Contract), eine *EAP-Success*-Nachricht vom Authenticator an der Peer gesendet. Damit ist die gesamte EAP-Authentifizierung abgeschlossen und der Peer hat die Autorisierung zum Netzwerkzugang erhalten.

Das von der BCA-Methode erzeugte Schlüsselmaterial (Master Session Key; MSK) kann nun von der darunter liegenden Protokollebene (bzw. 802.11) dazu verwendet werden die Netzwerkverbindung abzusichern (siehe dazu 3.1.13 *Erzeugung des EAP-Methoden Schlüsselmaterials*). Die Art der Absicherung ist dabei abhängig von der darunter liegenden Protokollebene.

Im Fall von 802.11 (WLAN) wird das Schlüsselmaterial zur Verschlüsselung der Datenübertragung eingesetzt.

Somit ist sichergestellt, dass selbst bei einer Weiterleitung der EAP-Authentifizierungsdaten zwischen dem (echten) Peer, einem Angreifer und dem (echten) Authenticator, es dem Angreifer nicht möglich ist auf das Netzwerk zuzugreifen.

3.1.11.1 Inital-Nachricht (InitMsg)

Die erste Nachricht der EAP-BCA-Methode und Schritt 3 im Ablauf ist die Inital-Nachricht (InitMsg; siehe Sequenzdiagramm, Abbildung 3.2).

In dieser Nachricht wird ein Zeitstempel (`timestamp`), der öffentliche Schlüssel der CA (`caPublicKey`), der öffentliche Schlüssel des Authenticators (`authPublicKey`) die zugehörige Schlüssel-Autorisierungs-Signatur (`authPublicKeySign`), der öffentliche ECDH-Schlüssel des Authenticators (`ecdhAuthPublicKey`) und die Authenticator-Authentifikation-Signatur (`authAuthenticationSign`) übertragen.

Anhand des öffentlichen Schlüssels des Authenticators (`authPublicKey`) ist der Authenticator identifizierbar.

Der Zeitstempel (`timestamp`) in Verbindung mit dem Hashwert des (einmaligen) öffentlichen ECDH(E)-Schlüssels des Authenticators (`ecdhAuthPublicKey`) dient der Sicherheit gegen Replay-Angriffe (siehe dazu 3.1.2 *Zeitstempel und Zufallszahlen*).

Der neu erzeugte öffentliche ECDH(E)-Schlüssel des Authenticators (`ecdhAuthPublicKey`) wird zum ECDH-Schlüsseltausch benötigt.

Die ECDSA-Signatur `authAuthenticationSign` wird zur Authentifizierung des Authenticators und zum Schutz der Integrität des ECDH(E)-Schlüssels des Authenticators eingesetzt.

Die Struktur der Inital-Nachricht (InitMsg) ist nachfolgend in Pseudocode beschrieben:

```
type InitMsg struct {
    timestamp                   Timestamp
    caPublicKey                 ECDSAPublicKey
    authPublicKey               ECDSAPublicKey
    authPublicKeySign           ECDSASignature
    ecdhAuthPublicKey           ECDHPublicKey
    authAuthenticationSign      ECDSASignature
}
```

Die Inital-Nachricht (InitMsg) hat eine feste Größe von 328 Bytes (Type-Data-Größe: 329 Byte; EAP-Nachrichten-Größe: 341 Byte; siehe 3.1.12 *Allgemeiner-Aufbau der EAP-BCA Nachrichten*).

Der nachfolgende Pseudocode zeigt die Erstellung der Inital-Nachricht `initMsg` im Authenticator:

```
timestamp           := timeNow()
ecdhAuthPrivateKey  := ecdhCreatePrivateKey()
ecdhAuthPublicKey   := ecdhCreatePublicKey(ecdhAuthPrivateKey)
ecdhAuthKeyHash     := hashDHKey(ecdhAuthPublicKey)


initMsg := InitMsg{
    timestamp,
    caPublicKey,
    authPublicKey,
    authPublicKeySign,
    ecdhAuthPublicKey,
    ecdsaSign("EAP-BCA-Authenticator-Auth" ∘ timestamp ∘ ecdhAuthPublicKey,
            authPrivateKey)
}
```

Der Authenticator sendet die Inital-Nachricht `initMsg` an den Peer und dieser verarbeitet die empfangene Nachricht.

Der nächste Pseudocode-Abschnitt zeigt die Verarbeitung der Inital-Nachricht `initMsg` im Peer:

```
timestamp           := initMsg.timestamp
caPublicKey         := initMsg.caPublicKey
authPublicKey       := initMsg.authPublicKey
authPublicKeySign   := initMsg.authPublicKeySign
nowtimestamp        := timeNow();

if ((timestamp + TimestampMaxDelta) < nowtimestamp ||
    (timestamp - TimestampMaxDelta) > nowtimestamp)
    error();

if (caKeyHash != hashDSAKey(caPublicKey))
    error()

authPublicKeySignCheck := ecdsaValidate(authPublicKey,
```

```
                                        authPublicKeySign,
                                        caPublicKey)

if (!authPublicKeySignCheck)
    error()


authAuthenticationSignCheck := ecdsaValidate(
        "EAP-BCA-Authenticator-Auth" ∘ timestamp ∘ initMsg.ecdhAuthPublicKey,
        initMsg.authAuthenticationSign,
        authPublicKey)

if (!authAuthenticationSignCheck)
    error()
```

3.1.11.2 Authentifizierungs-Nachricht (AuthMsg)

Die zweite und letzte Nachricht der EAP-BCA-Methode und Schritt 4 im Ablauf ist die Authentifizierungs-Nachricht (AuthMsg; siehe Sequenzdiagramm, Abbildung 3.2).

In dieser Nachricht wird der öffentliche ECDH(E)-Schlüssel des Peer (`ecdhPeerPublicKey`), der neu (zur Public-Key-Verschlüsselung) erzeugte öffentliche EC(DSA)-Schlüssel des Peer (`ecEncPublicKey`) und ein Chiffretext übertragen. Der Chiffretext enthält den öffentlichen Schlüssel des Peer (`peerPublicKey`) und die Peer-Authentifikation-Signatur (`peerAuthenticationSign`) in AEADCipher verschlüsselter Form.

Der neu erzeugte öffentliche ECDH(E)-Schlüssel des Peer (`ecdhPeerPublicKey`) wird zum ECDH-Schlüsseltausch benötigt.

Der Hashwert des öffentlichen ECDH(E)-Schlüssels des Peer (entspricht einer einmaligen Zufallszahl), dient in Kombination mit dem vom Authenticator erhaltenen Zeitstempel (`timestamp`), der Sicherheit gegen Replay-Angriffe (siehe dazu 3.1.2 *Zeitstempel und Zufallszahlen*).

Der neu erzeugte öffentliche EC(DSA)-Schlüssel des Peer (`ecEncPublicKey`) dient mit dem öffentlichen Schlüssel des Authenticators (`authPublicKey`) der Public-Key-Verschlüsselung. So das, nur der echte Authenticator der über den privaten Schlüssel des Authenticators (`authPrivateKey`) verfügt, in der Lage ist, den AEADCipher verschlüsselten Chiffretext zu entschlüsseln.

Damit wird die Vertraulichkeit, des öffentlichen Schlüssels des Peer (`peerPublicKey`) und der Peer-Authentifikation-Signatur (`peerAuthenticationSign`), gegenüber des echten Authenticators sichergestellt.

Die Hautaufgabe der Nachricht ist die Authentifikation des Peer gegenüber einem Authenticator durch die Peer-Authentifikation-Signatur (`peerAuthenticationSign`; siehe dazu 3.1.8.5 *`ecdsaValidate`-Methode*).

Der Hashwert, der aus dem öffentlichen ECDSA-Schlüssel des Peer erzeugt werden kann, dient als Identifikationsmerkmal des Peer und ermöglicht in Verbindung mit dem AAA-Contract die Autorisierung des Peer (siehe 3.2.1 *AAA-Contract*).

Die Struktur der Authentifizierungs-Nachricht (AuthMsg) ist nachfolgend in Pseudocode

beschrieben:

```
type AuthCipherPartMsg struct {
    peerPublicKey           ECDSAPublicKey
    peerAuthenticationSign  ECDSASignature
}

type AuthMsg struct {
    ecdhPeerPublicKey       ECDHPublicKey
    ecEncPublicKey          ECDSAPublicKey
    ciphered                AEADCipher
}
```

Die Authentifizierungs-Nachricht (AuthMsg) hat eine feste Größe von 280 Bytes (Type-Data-Größe: 281 Byte; EAP-Nachrichten-Größe: 293 Byte; siehe 3.1.12 *Allgemeiner-Aufbau der EAP-BCA Nachrichten*).

Der nachfolgende Pseudocode zeigt die Erstellung der Authentifizierungs-Nachricht `AuthMsg` im Peer:

```
ecdhPeerPrivateKey := ecdhCreatePrivateKey()
ecdhPeerPublicKey  := ecdhCreatePublicKey(ecdhPeerPrivateKey)
ecdhPeerKeyHash    := hashDHKey(ecdhPeerPublicKey)

sharedKey := ecdhGenerateKey(ecdhPeerPrivateKey, initMsg.ecdhAuthPublicKey)

ecEncPrivateKey := ecdsaCreatePrivateKey()
ecEncPublicKey  := ecdsaCreatePublicKey(ecEncPrivateKey)
ecEncSharedKey  := ecdhGenerateKeyByDSAKeys(ecEncPrivateKey, authPublicKey)

masterSecret := prf(
        sharedKey ∘ ecEncSharedKey,
        "master secret",
        ecdhPeerKeyHash ∘ ecdhAuthKeyHash,
        MasterSecretLength)

keyBlock := prf(
        masterSecret,
        "key expansion",
        ecdhAuthKeyHash ∘ ecdhPeerKeyHash,
        KeyBlockLength)

cipherParameters := CipherParameters{
    keyBlock[0:CipherKeyLength],
    keyBlock[CipherKeyLength:(CipherKeyLength + CipherSaltLength)]
}

authCipherPartMsg := AuthCipherPartMsg{
    peerPublicKey,
    ecdsaSign("EAP-BCA-Peer-Auth" ∘ timestamp ∘ sharedKey, peerPrivateKey)
}

authMsg := AuthMsg{
    ecdhPeerPublicKey,
    ecEncPublicKey,
    cipherEncryption(authCipherPartMsg, cipherParameters)
}
```

Der Peer sendet die Authentifizierungs-Nachricht `authMsg` an den Authenticator und dieser verarbeitet die empfangene Nachricht.

Der nächste Pseudocode-Abschnitt zeigt die Verarbeitung der Authentifizierungs-

Nachricht `authMsg` im Authenticator:

```
ecdhPeerKeyHash := hashDHKey(authMsg.ecdhPeerPublicKey)
sharedKey       := ecdhGenerateKey(ecdhAuthPrivateKey, authMsg.ecdhPeerPublicKey)
ecEncSharedKey  := ecdhGenerateKeyByDSAKeys(authPrivateKey,
                                            authMsg.ecEncPublicKey)

// START: same code part as in the creation of the authMsg
masterSecret := prf(
        sharedKey ∘ ecEncSharedKey,
        "master secret",
        ecdhPeerKeyHash ∘ ecdhAuthKeyHash,
        MasterSecretLength)

keyBlock := prf(masterSecret,
                "key expansion",
                ecdhAuthKeyHash ∘ ecdhPeerKeyHash,
                KeyBlockLength)

cipherParameters := CipherParameters{
    keyBlock[0:CipherKeyLength],
    keyBlock[CipherKeyLength:(CipherKeyLength + CipherSaltLength)]
}
// END: same code part

authCipherPartMsg := cipherDecryption(authMsg.ciphered, cipherParameters)

peerPublicKey := authCipherPartMsg.peerPublicKey

peerAuthenticationSignCheck := ecdsaValidate(
        "EAP-BCA-Peer-Auth" ∘ timestamp ∘ sharedKey,
        authCipherPartMsg.peerAuthenticationSign,
        peerPublicKey)

if (!peerAuthenticationSignCheck)
    error()

peerKeyHash := hashDSAKey(peerPublicKey)
```

Nach erfolgreicher Authentifizierung des Peer, kann der Authenticator sicher sein, den korrekten Hashwert des öffentlichen Schlüssels des Peer (`peerKeyHash`) zu kennen.

Dieser Hashwert dient als Identifikationsmerkmal des Peer, welchen an diesem Punkt zur Autorisation mit dem AAA-Contract verwendet wird (siehe 3.2.1 *AAA-Contract*).

Wenn der AAA-Contract die Autorisierung als erfolgreich ansieht, sendet der Authenticator dem Peer eine *EAP-Success*-Nachricht und öffnet für den Peer den Netzwerkzugang.

Sollte der AAA-Contract für die Autorisierung einen Fehlschlag zurückgeben, sendet der Authenticator dem Peer eine *EAP-Failure*-Nachricht und trennt die Verbindung.

Zur Absicherung des Netzwerkverbindung kann ein vor der BCA-Methode erzeugtes Schlüsselmaterial (Master Session Key; MSK) eingesetzt werden (siehe dazu 3.1.13 *Erzeugung des EAP-Methoden Schlüsselmaterials*).

3.1.12 Allgemeiner-Aufbau der EAP-BCA-Nachrichten

Um die unterschiedlichen EAP-BCA-Nachrichten in die EAP-Daten (*Type-Data* bzw. *Vendor-Data*) eines EAP-Pakets zu packen ist eine Container (`EapBcaData`) vorgesehen.

Dieser Container enthält den Typ der EAP-BCA-Nachricht und den Inhalt der EAP-BCA-Nachricht.

Die Struktur des allgemeinen Aufbaus der EAP-BCA-Nachrichten ist nachfolgend in Pseudocode beschrieben:

```
const {
    BCA_MSG_TYPE_INIT = 0
    BCA_MSG_TYPE_AUTH = 1
}

type BCAMessageType uint8

type EapBcaData struct {
    messageType     BCAMessageType
    messageData     select (.messageType) {
        case BCA_MSG_TYPE_INIT: InitMsg
        case BCA_MSG_TYPE_AUTH: AuthMsg
    }
}


eapBcaData := EapBcaData{
    messageType,
    select (messageType) {
        case BCA_MSG_TYPE_INIT: initMsg
        case BCA_MSG_TYPE_AUTH: authMsg
    }
}
```

Die Größe der EAP-Daten (*Type-Data* bzw. *Vendor-Data*) entspricht dabei der Größe der jeweiligen BCA-Nachricht plus ein Byte für die Nachrichten-Typ- (`messageType`) Informationen.

Da die EAP-BCA-Methode als *Expanded Type* implementiert ist, kommen zum vollständigen EAP-Paket noch einmal weitere 12 Byte zu den EAP-Daten hinzu (siehe dazu 1.4 *Extensible Authentication Protocol (EAP)*).

3.1.13 Erzeugung des EAP-Methoden Schlüsselmaterials

Eine EAP-Methoden die der Authentifizierung dient kann Schlüsselmaterial herleiten.

Dieses Schlüsselmaterial besteht aus einem *Master Session Key* (MSK) und einem *Extended Master Session Key* (EMSK) (für mehr Informationen siehe 3.1.1 *Terminologie*).

Die Herleitung des Schlüsselmaterials ist im nachfolgenden Pseudocode gezeigt:

```
mskBlock := prf(masterSecret,
        "client EAP encryption",
        ecdhPeerPublicKey ∘ ecdhAuthPublicKey,
        MSKBlockLength)

msk  := mskBlock[0:MSKLength]
emsk := mskBlock[MSKLength:(MSKLength + EMSKLength)]
```

3.1.14 Erzeugung der EAP-Session-ID

Zur Zuordnung einer Sitzung verfügt eine EAP-Methode über eine Methode die eine *Session-ID* zurückgibt.

Die Erzeugung dieser *Session-ID* (`sessionId`) wird im nachfolgenden Pseudocode gezeigt:

```
EAPTypeExtended := BCAVendorId ∘ BCAVendorType

sessionId := EAPTypeExtended ∘ timestamp ∘ ecdhAuthPublicKey ∘ ecdhPeerPublicKey
```

BCAVendorId und *BCAVendorType* sind dabei die *Vendor-Id* und der *Vendor-Type* der EAP-BCA-Methode (siehe dazu 1.4.2 *Expanded Types*) und zu Testzwecken in dieser Arbeit frei gewählt.

3.1.15 Sicherheitsansprüchen

Dieser Abschnitt befasst sich mit Sicherheitsansprüchen wie es laut EAP-Standard von jeder Spezifikation einer EAP-Methode verlangt wird.

Die dabei zu behandelnden Punkte stammen aus der Deklaration im Kapitel 7.2 *Security Claims* des EAP-Standards (RFC3748; siehe Anhang A.2).

Die Sicherheitsansprüche (im Englischen: Security Claims) von EAP-BCA sind die folgenden:

```
Auth. mechanism:           ECDSA keys
Ciphersuite negotiation:   No
Mutual authentication:     Yes
Integrity protection:      Yes
Replay protection:         Yes
Confidentiality:           Yes
Key derivation:            Yes
Key strength:              128
Dictionary attack prot.:   Yes
Fast reconnect:            No
Crypt. binding:            N/A
Session independence:      Yes
Fragmentation:             No
Channel binding:           No
```

3.2 Blockchain Triple-A-System Backend

Der folgende Abschnitt behandelt das Konzept, des in der Blockchain laufenden Triple-A-System-Backends (AAA-Backends).

Die Voruntersuchungen haben gezeigt das die Blockchain Ethereum für diese Arbeit, das Mittel der Wahl ist (siehe dazu 2.1 *Blockchain*) und somit die Konzeption und Realisierung des AAA-Backends als Contract in Ethereum erfolgt.

3.2.1 AAA-Contract

Zu einfachen demonstration des gesamten Blockchain-basierten Triple-A-Systems, wird nachfolgend eine Contract-Struktur konzipiert, mit der ein rudimentäres AAA-Backend in der Blockchain Ethereum etabliert werden kann.

Welche Aufgaben soll ein AAA-Backend-Contract (AAA-Contract) erfüllen?

Der AAA-Contract soll in der Lage seinen mit Hilfe des EAP-BCA-Protokolls (siehe 3.1 *EAP-Blockchain-Authorisation Protokoll*) eine Authentifizierung und Autorisierung eines Nutzers (Nutzergerätes) durchzuführen bzw. Nutzer verwalten zu können.

Neben Nutzern, soll es mit dem AAA-Contract möglich sein, Administratoren und Authenticators zu verwalten.

Und es soll es mit dem AAA-Contract möglich sein, Accounting-Informationen der Nutzer in der Blockchain zu speichern.

Um dies alles zu ermöglichen muss der AAA-Contract einige Informationen speichern und Methoden bereitstellen um auf diese Informationen zuzugreifen.

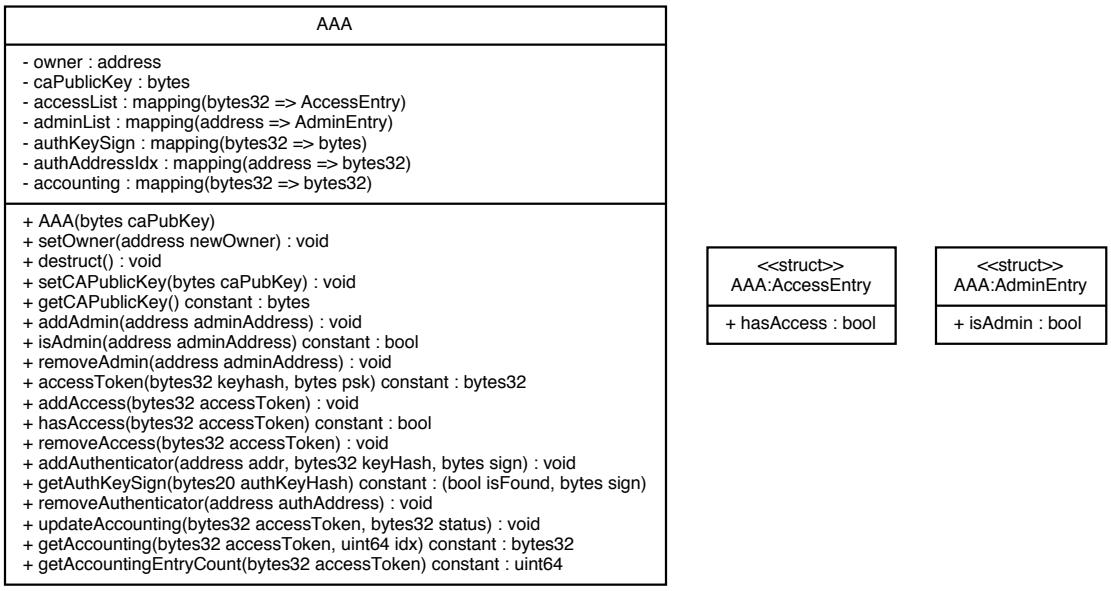


Abbildung 3.3: *Klassendiagramm: Triple-A-System Backend Contract*

Abbildung 3.3 zeigt den AAA-Contract mit all seinen Attributen und Methoden, welche nachfolgen genauer Beschrieben werden.

3.2.1.1 Interne Strukturen

```
struct AccessEntry {
    bool hasAccess;
}

struct AdminEntry {
    bool isAdmin;
}
```

Die beiden internen Strukturen `AccessEntry` und `AdminEntry` dienen der bessern lesbarkeit des Programmcodes, denn:

```
require(adminList[msg.sender].isAdmin == true);
```

... ist besser verständlich als:

```
require(adminList[msg.sender] == true);
```

Und ermöglicht es, schnell weitere Argumente für Bedingungen zu den Strukturen hinzuzufügen.

Ein Beispiel dazu wäre ein Ablaufdatum für Administrationsrechte oder feste Zeiten, zu denen ein Nutzer Netzwerkzugriff erhält.

Interne Strukturen zu diesen Beispiel-Erweiterungen:

```
struct AccessEntry {
    bool   hasAccess;
    uint32 startDayTime;
    uint32 stopDayTime;
}

struct AdminEntry {
    bool   isAdmin;
    uint32 expirationTimestamp;
}
```

3.2.1.2 Interne States

```
    address  owner
    bytes    caPublicKey

    mapping(bytes32 => AccessEntry) accessList
    mapping(address => AdminEntry)  adminList

    mapping(bytes32 => bytes)       authKeySign
    mapping(address => bytes32)     authAddressIdx

    mapping(bytes32 => bytes32)     accounting
```

Beschreibung:

owner
: Ethereum-Adresse des Contract Eigentümers.

caPublicKey
: Öffentlicher ECDSA-Schlüssel der CA.

accessList
: Mapping von *accessToken* zu *AccessEntry*, der Aufschluss darüber gibt, ob ein Peer

mit dem angegebenen *accessToken*, Zugriff auf das Netzwerk hat (und **optional** zu welchen Bedingungen / nicht in der gezeigten Implementierung enthalten; siehe 3.2.1.1 *Interne Strukturen*). Entspricht einer Nutzer-Liste.

adminList
Mapping einer Ethereum-Adresse (eines Administrators) zu einem *AdminEntry*, der Aufschluss darüber gibt ob ein Ethereum-Account mit seiner Ethereum-Adresse, Administrativen-Zugriff auf den AAA-Contract hat (und **optional** zu welchen Bedingungen / nicht in der gezeigten Implementierung enthalten; siehe 3.2.1.1 *Interne Strukturen*). Damit entspricht dieses Mapping einer Liste mit Administratoren.

authKeySign
Mapping eines Hashwerts des öffentlichen ECDSA-Schlüssels eines Authenticators zu dessen Schlüssel-Signatur. Entspricht einer Liste mit autorisierten Authenticators (Authenticators-Liste).

authAddressIdx
Mapping einer Ethereum-Adresse eines Authenticator zu dem Hashwerts von dessen öffentlichen ECDSA-Schlüssels. Entspricht einem Index zu der Authenticators-Liste.

accounting
Mapping aus einer Kombination aus *accessToken* und Accounting-Index zu einer Accounting-Nachricht. Entspricht indexierten Accountingdaten zu einzelnen Nutzern. Zu jeder Kombination aus *accessToken* und Accounting-Index gibt es einen Accounting-Nachrichten-Eintrag.

3.2.1.3 Initialisierung des Contracts

Bei der Erstellung und der damit einhergehenden Initialisierung des AAA-Contracts wird mit der Constructor-Methode gleich der öffentliche ECDSA-Schlüssel der CA (`caPublicKey`) für diesen AAA-Contract als Parameter übergeben bzw. übertragen.

```
function AAA(bytes caPubKey)
```

3.2.1.4 Contract-Verwaltungs-Methoden

Um als Ersteller und Eigentümer des AAA-Contracts, diesen Verwalten zu können, werden spezielle Methoden bereitgestellt.

Nur der Eigentümer (*owner*) des AAA-Contracts kann diese Methoden verwenden. Um dies sicherzustellen, werden die Methoden beim Aufrufen von einer anderen Ethereum-Adresse als der des Eigentümers, direkt zu Begin, mit einem Fehler beendet.

Contract-Verwaltungs-Methoden:

```
function setOwner(address newOwner)
function destruct()
function setCAPublicKey(bytes caPubKey)
```

Beschreibung:

setOwner(address newOwner)
Übergibt den AAA-Contract an den Blockchain-Nutzer mir der Ethereum-Adresse `newOwner`. Welcher danach der neue Eigentümer (*owner*) des AAA-Contracts ist.

destruct()
Terminiert den AAA-Contract. Der Contract ist danach nicht mehr verwendbar.

setCAPublicKey(bytes caPubKey)
Ersetzt den bisherigen öffentlichen ECDSA-Schlüssel der CA durch den als Parameter (`caPubKey`) angegebenen.

3.2.1.5 Administrator-Management des AAA-Contracts

Die nachfolgenden Methoden dienen dem Management der Administratoren.

Nur die Administratoren und der Eigentümer (*owner*) des AAA-Contracts können diese Methoden verwenden. Um dies sicherzustellen, werden die Methoden beim Aufrufen von einer anderen Ethereum-Adresse als der eines Administrators oder des Eigentümers, direkt zu Begin, mit einem Fehler beendet.

Administrator-Management-Methoden:

```
function addAdmin(address adminAddress)
function isAdmin(address adminAddress) constant returns (bool isadmin)
function removeAdmin(address adminAddress)
```

Beschreibung:

addAdmin(address adminAddress)
Fügt einen neuen Administrator mit der angegebenen Ethereum-Adresse (`adminAddress`) zu der Liste der Administratoren hinzu.

isAdmin(address adminAddress)
Die Lokal-aufrufbare Methode gibt `true` zurück falls es sich bei der übergebenen Ethereum-Adresse (`adminAddress`) um einen Administrator handelt, falls nicht wird `false` zurückgegeben.

removeAdmin(address adminAddress)
Entfernt den Administrator mit der angegebenen Ethereum-Adresse (`adminAddress`) aus der Liste der Administratoren.

3.2.1.6 Anonymisierung der Nutzerinformationen

Zum Schutz der Identität der Nutzer werden die Hashwerte der öffentlichen ECDSA-Schlüssel der Peers mithilfe des sich pro AAA-Contract unterscheidenden Pre-Shared-Keys (PSKs), in Form eines Zugriffmerksmals (*accessToken*) anonymisiert.

Somit ist es, trots der Nutzung des gleichen öffentlichen ECDSA-Schlüssels eines Peers in unterschiedlichen AAA-Contracts (mit unterschiedlichen PSKs), nicht möglich die daraus resultierenden unterschiedlichen *accessTokens* mit einander in Verbindung zu bringen.

Genauso ist es nicht möglich, ohne den PSK einen Zusammenhang zwischen einem *accessToken* und einem Hashwert eines öffentlichen ECDSA-Schlüssels eines Peer herzustellen.

Für die Erstellung eines *accessToken* ist eine Methode im AAA-Contract vorgesehen:

```
function accessToken(bytes32 keyhash, bytes psk) constant
        returns (bytes32 token)
```

Die `accessToken`-Methode erzeugt dabei den *accessToken* aus dem Hashwert des öffentlichen ECDSA-Schlüssels des Peer `keyhash` und dem PSK `psk` und gibt diesen zurück.

3.2.1.7 Nutzer-Management

Die nachfolgenden Methoden dienen dem Management der Nutzer (bzw. der Nutzergeräte) und dessen Authentifizierung und Autorisierung.

Nur die Administratoren und der Eigentümer (*owner*) des AAA-Contracts können diese Methoden verwenden. Um dies sicherzustellen, werden die Methoden beim Aufrufen von einer anderen Ethereum-Adresse als der eines Administrators oder des Eigentümers, direkt zu Begin, mit einem Fehler beendet.

Nutzer-Management-Methoden:

```
function addAccess(bytes32 accessToken)
function hasAccess(bytes32 accessToken) constant returns (bool bHasAccess)
function removeAccess(bytes32 accessToken)
```

Beschreibung:

addAccess(bytes32 accessToken)
: Fügt einen neuen Nutzer mit dem angegebenen *accessToken* zu der Liste der Nutzer hinzu.

hasAccess(bytes32 accessToken)
: Die Lokal-aufrufbare Methode gibt `true` zurück falls es sich bei dem übergebenen *accessToken* um einen Autorisierten Nutzer handelt. Also einem Nutzer der in der Liste der Nutzer vorhanden ist und somit der Zugriff aus das Netzwerk gestattet wird. Falls nicht wird `false` zurückgegeben.

removeAccess(bytes32 accessToken)
: Entfernt den Nutzer mit dem angegebenen *accessToken* aus der Liste der Nutzer und somit die Autorisierung zur Netzwerknutzung.

3.2.1.8 Authenticator-Management

Die nachfolgenden Methoden dienen dem Management der Authenticators, bzw. dem hinzufügen und entfernen von autorisierten Authenticators.

Das hinzufügen eines neuen Authenticators ist dabei Administratoren vorbehalten die im Besitz des privaten ECDSA-Schlüssels der CA sind bzw. über diesen Verfügen können, um mit diesem den öffentlichen ECDSA-Schlüssel des Authenticators signieren zu können.

Nur die Administratoren und der Eigentümer (*owner*) des AAA-Contracts können diese Methoden verwenden. Um dies sicherzustellen, werden die Methoden beim Aufrufen von einer anderen Ethereum-Adresse als der eines Administrators oder des Eigentümers, direkt zu Begin, mit einem Fehler beendet.

```
function addAuthenticator(address authAddress,
                          bytes32 authKeyHash,
                          bytes keySign)
function removeAuthenticator(address authAddress)
```

Beschreibung:

addAuthenticator(...)
: Fügt einen neuen Authenticator mit der angegebenen Ethereum-Adresse `authAddress`, dem Hashwert des öffentlichen ECDSA-Schlüssels des Authenticators `authKeyHash` und der zugehörigen Schlüssel-Signatur `keySign` zu der Liste der Authenticators hinzu.

removeAuthenticator(address authAddress)
: Entfernt den Authenticator mit der Ethereum-Adresse `authAddress`aus der Liste der Authenticators.

3.2.1.9 Authenticator-Informations-Methoden

Die nachfolgenden Methoden sind beide Lokal-Ausführbar und geben wichtige Informationen zurück, wie die Schlüssel-Signatur eines Authenticators oder den öffentlichen ECDSA-Schlüssel der CA. Diese Informationen sind für einen Peer von Bedeutung, der die Autorisierung eines Authenticators überprüfen muss.

```
function getAuthKeySign(bytes32 authKeyHash) constant
        returns (bool isFound, bytes sign)
function getCAPublicKey() constant returns (bytes)
```

Beschreibung:

getAuthKeySign(bytes32 authKeyHash)
: Die Lokal-aufrufbare Methode gibt `true` und die Schlüssel-Signatur des Authenticators mit dem Hashwert des öffentlichen ECDSA-Schlüssels `authKeyHash` zurück, wenn sich dieser in der Liste der Authenticators befindet. Falls der angegebene Authenticator nicht in der Liste der Authenticators aufgeführt ist und somit nicht Autorisiert ist, wird `false` und `null` zurückgegeben.

getCAPublicKey()
: Die Lokal-aufrufbare Methode gibt den öffentlichen ECDSA-Schlüssel der CA zurück.

3.2.1.10 Accounting

Für das Accounting in der Blockchain stellt der AAA-Contract spezielle Methoden bereit.

Die Nutzung dieser Methoden ist den Authenticators und den Administratoren vorbehalten. Um sicher zu stellen das nur diese Gruppe von Blockchain-Nutzern die Methoden verwenden können, wird beim Aufrufen einer der folgenden Methoden von einer anderen Ethereum-Adresse, als von der eines sich in der Liste befindenden Authenticators oder Administrators, die aufgerufene Methode direkt zu Begin mit einem Fehler beendet.

Accounting-Methoden:

```
function updateAccounting(bytes32 accessToken, bytes32 status)
function getAccounting(bytes32 accessToken, uint64 index) constant
        returns (bytes32)
function getAccountingEntryCount(bytes32 accessToken) constant
        returns (uint64)
```

Beschreibung:

updateAccounting(bytes32 accessToken, bytes32 status)
: Sendet und speichert die Accounting-Nachricht `status` für den Nutzer mit dem angegebenen *accessToken* als neuen Accounting-Eintrag für den Nutzer im AAA-Contract (und somit in der Blockchain).

getAccounting(bytes32 accessToken, uint64 index)
: Die Lokal-aufrufbare Methode gibt die Accounting-Nachricht mit der (Durchlauf-/Index-) Nummer `index` des Nutzers mit dem angegebenen *accessToken* zurück. Fall diese Accounting-Nachricht noch nicht existiert werden NULL-Daten (32 Bytes die alle den Wert 0 haben) zurück gegeben.

getAccountingEntryCount(bytes32 accessToken)
: Die Lokal-aufrufbare Methode gibt die Anzahl der gespeicherten Accounting-Nachrichten für den Nutzer mit dem angegebenen *accessToken* zurück.

Um ein bedarfsgerechtes Accounting zu ermöglichen, sollen für verschiedene (denkbare) Ereignisse auch unterschiedliche Accounting-Nachrichten gespeichert werden können. Die Accounting-Nachrichten enthalten zur Unterscheidung eine Typen-Angabe (`messageType`). Durch diese Angabe sind unterschiedliche Accounting-Nachrichten-Typen unterscheidbar.

Die Struktur des Accounting-Nachrichten-Containers:

```
type AccountingMsg struct {
    timestamp       uint32
    authKeyHashPart [4]byte
    ciphered        []byte
}
```

Beschreibung:

`timestamp`
: (Unix-)Zeitstempel der Accounting-Nachricht, der die Anzahl der Sekunden die seit dem 1.1.1970 um 00:00 Uhr (UTC) vergangen sind, angibt.

`authKeyHashPart`
: Ersten 4 Bytes des Hashwerts des öffentlichen ECDSA-Schlüssels des Authenticators, der die Authentifizierung durchgeführt hat.

`ciphered`
: Container der die Daten des verschlüsselten Accounting-Nachrichtenteils beinhaltet.

Die eigentliche Accounting-Nachricht ist Verschlüsselt. Nur der Zeitstempel und die ersten 4 Bytes des Hashwerts des öffentlichen ECDSA-Schlüssels des Authenticators (`authKeyHash[0:4]`) sind unverschlüsselt, da diese Daten ohnehin über die Blockchain-Transaktion die zum Ausführen der `updateAccounting`-Methode gesendet wurde, ersichtlich sind.

Der verschlüsselte Teil der Accounting-Nachricht hat folgende Grundstruktur:

```
const {
    BC_ACCOUNTING_MSG_TYPE_DISCONNECT = 1
}

type AccountingCipheredMsg struct {
    messageType     uint8
    messageData     select (.messageType) {
        case BC_ACCOUNTING_MSG_TYPE_DISCONNECT: AccountingDisconnectMsg
    }
}
```

Disconnect-Nachricht Die Disconnect-Nachricht wird von einem Access-Point gesendet, wenn ein Nutzer die Verbindung mit dem Netzwerk trennt (Englisch: *disconnect*; vom Deutschen: *trennen*).

Eine Disconnect-Nachricht enthält die MAC-Adresse (Media-Access-Control-Adresse) der Schnittstelle (des Netzwerkadapters) mit der das Nutzergerät auf das Netzwerk zugreift (`macAddress`), die während der Verbindung insgesamt übertragenen Daten in Bytes (`rxTxBytes`) und die jeweils von dem Peer empfangene und gesendete Anzahl an Paketen (`rxPackets` und `txPackets`).

Pseudocode der Disconnect-Nachrichtenstruktur:

```
type AccountingDisconnectMsg struct {
    macAddress      [6]byte
    rxTxBytes       uint64
    rxPackets       uint32
    txPackets       uint32
}
```

Eine Accounting-Disconnect-Nachricht hat eine feste Größe von 31 Bytes.

3.3 Nutzungskonzept

Dieser Teil behandelt das Nutzungskonzept des Blockchain-basierten Triple-A-Systems.

Und soll die Fragen beantworten, wie ein Authenticator und eine Nutzergerät (Peer) eingerichtet wird und was dazu getan werden muss.

3.3.1 Konfiguration eines Authenticators

Die Konfiguration eines Authenticators bzw. eines WLAN-Access-Points für die Nutzung des Blockchain-basierten Triple-A-Systems besteht aus wenigen Konfigurationsparametern die unter 4.1.1.4 *Konfigurationsdaten* beschreiben werden.

Die gesamten Konfigurationsdaten können auf dem Rechnersystem des Administrators erzeugt und anschließend auf das Gerät kopiert werden.

Dabei müssen ein neuer privater ECDSA-Schlüssel und ein Ethereum-Account für den Authenticator angelegt werden. Es gilt dabei zu beachten, dass der Authenticator für das Blockchain-Accounting einen zumindest kleinen Etat in Ether, für das Zahlen der Gas-Kosten die beim Ausführen der `updateAccounting`-Methode anfallen, zur Verfügung gestellt bekommt.

Zur Konfiguration stellen Access-Points meist Remote-Zugriffe in Form eines Konsolen- und/oder Web-Interfaces bereit. Über diese lassen sich nun die Parameter einstellen.

Um den Authenticator als solchen zu autorisieren, muss der öffentliche ECDSA-Schlüssel (des für diesen Authenticator erzeugten privaten ECDSA-Schlüssels) mit dem privaten ECDSA-Schlüssel der CA signiert werden. Und diese Signatur, mit dem öffentlichen Schlüssel und der Ethereum-Adresse des Authenticators im AAA-Contract mit der Methode `addAuthenticator`, zu der Liste der autorisierten Authenticators hinzugefügt werden.

3.3.2 Gerätekonfiguration (Peer)

Nachfolgend werden zwei (primäre) Szenarien vorgestellt, wie ein Gerät eines Nutzers für den Netzwerkzugriff konfiguriert werden kann.

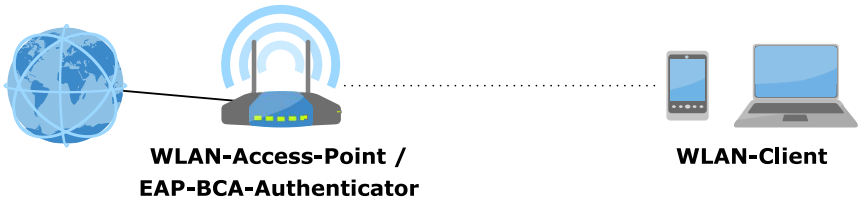
Die Nutzungsdaten (Netzwerk-Zugangsdaten) bestehend neben dem privaten ECDSA-Schlüssel des Peer (bzw. dem daraus abgeleiteten Hashwert des öffentlichen ECDSA-Schlüssels), aus der SSID des WLANs, der Netzwerk-ID (`bcNetworkId`) und dem Hashwert des öffentlichen ECDSA-Schlüssels der CA (`caKeyHash`).

3.3.2.1 Szenario I

Dieses Szenario behandelt denn Fall, dass das Nutzungsgerät über eine vollwertige Benutzerschnittstelle (im Englischen: User Interface; UI) verfügt, mit dem der Benutzer die Netzwerk-Zugangsdaten eingeben kann.

Zuerst muss hier unterschieden werden ob der Nutzer einen neuen oder bestehenden privaten ECDSA-Schlüssel (`peerPrivateKey`; somit auch `peerPublicKey` und `peerKeyHash`) für das Gerät verwenden möchte.

Bei der Nutzung eines neuen Schlüssels für das Gerät muss dieser vorher erzeugt werden. Hierfür eignet sich jeder (kryptografisch) sicherere Zufallszahlengenerator, der eine 32 Byte lange Zufallszahl erzeugen und hexadezimal Ausgeben kann.



Der Hashwert des verwendeten öffentlichen ECDSA-Schlüssels (`peerKeyHash`) muss der Nutzer nun an einen Administrator des Netzwerks zur Freischaltung weiter geben.

Die Weitergabe kann dabei direkt an einen Administrator erfolgen oder indirekt durch den Einsatz von Tools erfolgen, die diesen Schritt automatisieren.

Der Administrator oder das Tool nutzt nun den AAA-Contract um aus dem `peerKeyHash` einen *accessToken* zu erstellen und dieses dann mit der Methode `addAccess` für den Netzwerkzugriff zu Autorisieren.

Der Nutzer erhält die Nutzungsdaten entweder direkt von einem Administrator des Netzes oder indirekt, indem die Daten durch ein beliebiges Medium dem Nutzer bereitgestellt werden.

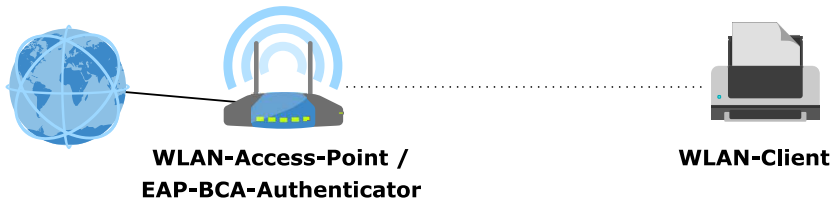
Die erhaltenen Nutzungsdaten werden nun von Nutzer (oder auch Administrator) im Nutzungsgerät eingegeben und dieses kann im Anschluss das Netzwerk nutzen.

Ein Beispiel der Nutzungsdaten bzw. einer Geräte-Konfiguration ist unter 4.1.2 *wpa_supplicant* gezeigt.

3.3.2.2 Szenario II

Dieses Szenario behandelt denn Fall, dass das Nutzungsgerät nur über **eingeschränkte** Benutzerschnittstelle verfügt, die **keine** vollwertige Tastatur oder einen Touchscreen bereitstellt. Somit muss der Benutzer zur Konfiguration des Netzwerkzugangs auf eine sehr minimalistische Eingabe- und Ausgabe-Schnittstelle zurückgreifen, um die (erste) Verbindung mit dem Netzwerk herstellen zu können.

Einige Beispiele für solche Geräte sind Netzwerk-Drucker, Multimedia-Geräte mit Internetanbindung, IoT- (IoT, englische Abkürzung für: Internet of Things; im Deutschen: Internet der Dinge) und SmartHome-Geräte, welche alle zum überwiegenden Teil keine guten Eingabemöglichkeiten bieten, um längere Zeichen- und Zahlenfolgen einzugeben.



Grundvoraussetzung an ein solches Gerät ist, dass dieses über eine WLAN-Schnittstelle und ein Display mit Eingabemöglichkeit verfügt, mit der es möglich ist eine SSID aus einer angezeigten Liste auszuwählen, eine kurze Zeichenfolge einzugeben und einen Hashwert (`peerKeyHash`) hexadezimal anzuzeigen.

Der Ablauf um das Gerät mit dem Netzwerk zu verbinden, ist der folgende:

Der Nutzer lässt den Hashwert des öffentlichen ECDSA-Schlüssels des Gerätes (`peerKeyHash`) anzeigen und liest diesen ab. Dieser Hashwert wird dann wie im ersten Szenario an einen Administrator zur Freischaltung weiter gegeben (siehe 3.3.2.1 *Szenario I*).

Nach der Freischaltung das *accessToken* im AAA-Contract startet der Nutzer im Gerät den Verbindungsaufbau und wählt die in den Nutzungsdaten angegebene SSID aus einer Liste mit SSIDs, der gefundenen WLANs aus. Im Anschluss gibt er *EAP-BCA* als Authentifizierungsmethode an und die sehr kurze `bcNetworkId` ein. Zum Abschluss vergleicht der Nutzer den angezeigten mit dem in den Nutzungsdaten angegebenen Hashwert des öffentlichen ECDSA-Schlüssels der CA (`caKeyHash`) und bestätigt diesen, wenn dieser übereinstimmt.

Nun kann sich das Gerät jederzeit mit dem Netzwerk verbinden und erhält Zugriff auf das Netzwerk, solange dessen Schlüssel im AAA-Contract des Netzes autorisiert ist.

3.3.3 Entziehung von Zugriffsrechten

Die Entziehung einer Nutzungsberechtigung ist für einen Administrator mit der `removeAccess`-Methode des AAA-Contracts sehr einfach und jederzeit möglich.

Kapitel 4

Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung des konzipierten Blockchain-basierten Triple-A-Systems.

Und somit, mit der Implementierung der EAP-Erweiterung BCA zur Authentifizierung über IEEE 802.1X (siehe 3.1 *EAP-Blockchain-Authorisation (EAP-BCA) Protokoll*) in eine WLAN-Access-Point- und WLAN-Client-Software, sowie der Implementierung des Blockchain-Accountings in der WLAN-Access-Point-Software und der Implementierung eines Blockchain-Smart-Contract der die Aufgabe des Triple-A-System-Backends erfüllt.

Die WLAN-Access-Point-Software *hostapd* und die WLAN-Client-Software *wpa_supplicant* werden dazu um die EAP-Methode BCA erweitert und in *hostapd* ein Blockchain-Accounting implementiert.

4.1 hostap

Das Open-Source-Projekt *hostap* ist mit seinen Programmen *hostapd* und *wpa_supplicant* Bestandteil des Linux-Projekts.

Der Quellcode von *hostapd* und *wpa_supplicant* ist in einem gemeinsamen Quellcode-Verzeichnis untergebracht und als Git-Repository unter `http://w1.fi/cgit/hostap/` verfügbar.

Des gesamte *hostap*-Projekt steht unter der BSD-Lizenz und der Programmcode von *hostapd* und *wpa_supplicant* zusammen umfasst 453 Quellcode-Dateien und 277 Header-Dateien mit zusammen über 11MB Quellcode.

hostapd und *wpa_supplicant* implementieren die in *RFC 4137* beschriebenen State-Machines für einen EAP-Authenticator und einen EAP-Peer (siehe Abbildung 4.1 und 4.2). Dies stellt ein erweiterbares Konzept mit einheitlichen Schnittstellen sicher und ermöglicht es so, beliebig EAP-Methoden zu *hostapd* und *wpa_supplicant* hinzuzufügen.

Sowohl *hostapd* als auch *wpa_supplicant* kommen im Betriebssystem *Debian* als Standard WLAN-Software zum Einsatz. Da *Debian* das Betriebssystem der angestrebten Implementierungsumgebung ist (siehe 2.4 *Implementierungsumgebung*), ist das Verwenden und die Erweiterung dieser beiden Programme um die EAP-BCA-Methode ein logische Konsequenz.

4.1.1 hostapd

Die Software *hostapd* ist ein User-Space-Daemon (Programme die mit Benutzerrechten im Hintergrund laufen) für WLAN-Access-Points und Authentifikations-Server. Dabei werden WPA (abgekürzt für *Wi-Fi Protected Access*) und WPA2, sowie alle gängigen EAP-Methoden unterstützt [1].

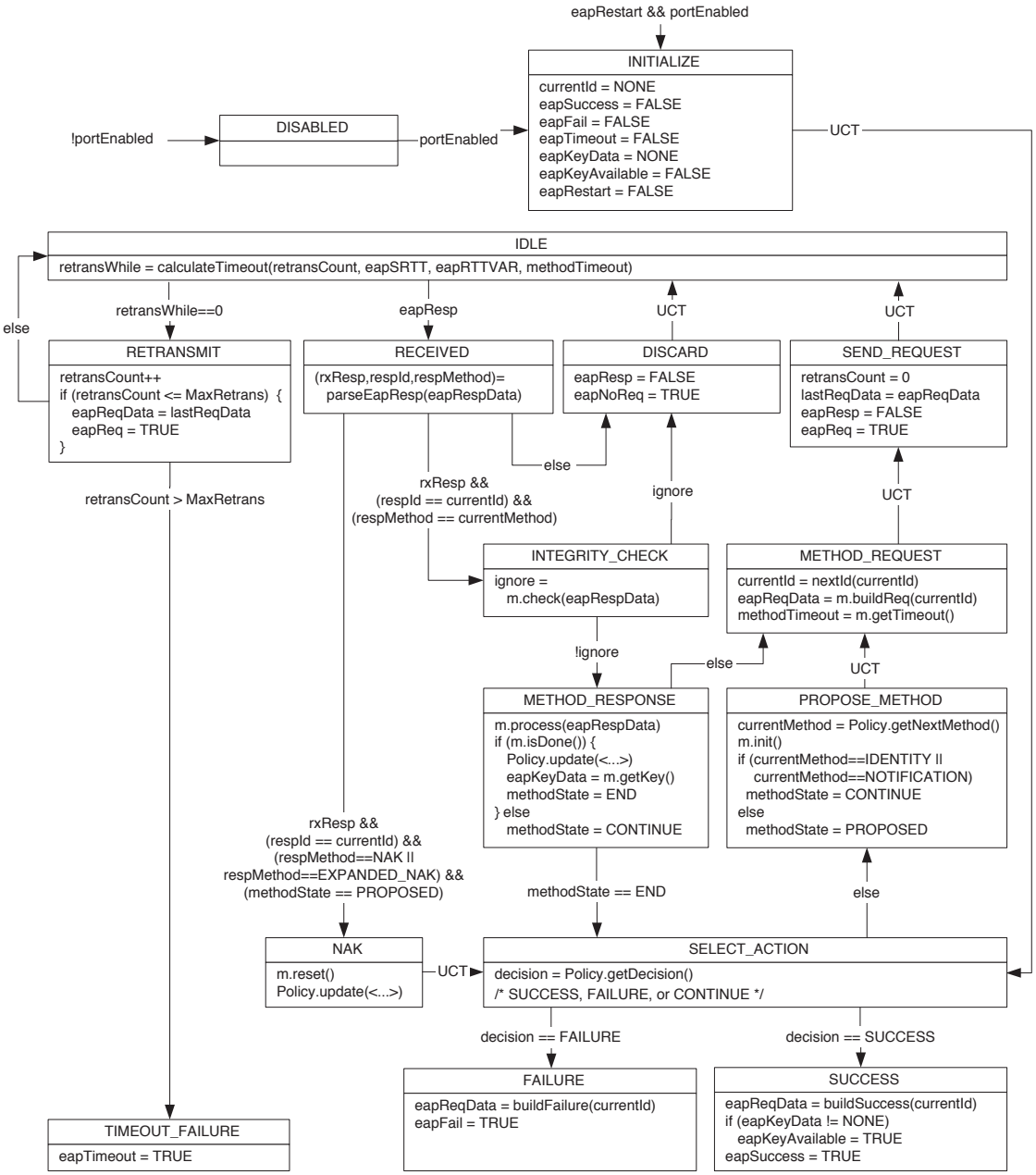


Abbildung 4.1: *EAP Stand-Alone Authenticator State-Machine [36]*

In Abbildung 4.1 ist ersichtlich, dass jede EAP-Methode im Authenticator eine Schnittstelle (im Englischen: Interface) aus Funktionen (Methoden) zur Verfügung stellen muss.

Die Schnittstellen-Methoden einer EAP-Methode (laut RFC 4137 State-Machine) eines EAP-Authenticators sind:

`init()`
: Initialisierung der EAP-Methode.

`buildReq(currentId)`
: Erzeugt den aktuellen EAP-Request der EAP-Methode.

`check(eapRespData)`
: Überprüft den eingegangenen EAP-Response und entscheidet ob dieser ignoriert werden soll.

`process(eapRespData)`
: Verarbeitet den eingegangenen EAP-Response.

`reset()`
: Setzt die internen Variablen der EAP-Methode zurück.

`getTimeout()`
: Gibt die maximale Wartezeit auf eine Antwort des Peers der EAP-Methode zurück.

`isDone()`
: Gibt `true` zurück falls die EAP-Methode abgeschlossen ist, ansonsten `false`.

`getKey()`
: Gibt das ausgehandelte Schlüsselmaterial zurück.

`isSuccess()`
: Gibt `true` zurück falls die Authentifizierung mit der EAP-Methode erfolgreich abgeschlossen wurde, ansonsten `false`.

Diese Schnittstelle ist für die EAP-BCA-Methode des Peer in der Quellcode-Datei *src/eap_peer/eap_bca.c* implementiert.

4.1.1.1 Blockchain-Kommunikation

Die Interaktion mit der Blockchain erfolgt über den Ethereum-Clienten *geth*.

Die Kommunikation mit *geth* erfolgt über eine Interprozesskommunikation (im Englischen: interprocess communication; IPC) mit einer Socket-Datei die *geth* erzeugt.

Beim Ausführen einer nicht-lokalen Methode des AAA-Contracts (in der Blockchain) muss der Methoden-Aufruf per Transaktion an den AAA-Contract gesendet werden. Zum Senden einer Transaktion bedarf es dem privaten (Ethereum-) Schlüssel des (Absendenden-) Accounts, mit dem die Transaktion signiert werden muss. Um den privaten (Ethereum-) Schlüssel für diesen Vorgang bereitzustellen, muss in *geth* für den Account der passenden *Passphrase* (Entschlüsselungs-Passwort) eingegeben werden, um den (nur verschlüsselt gespeicherten) privaten (Ethereum-) Schlüssels zu entschlüsseln (mit dem *geth* Funktions-Aufruf: `personal.unlockAccount`).

Was heißt das zum Ausführen einer nicht-lokalen Methode eines Contracts mit *geth*, immer die Ethereum-Adresse des Absenders und der passenden *Passphrase* des Accounts benötigt wird.

Weshalb diese bei der Konfiguration als Parameter angegeben werden müssen.

4.1.1.2 Quellcode

Zur Erweiterung der Funktionalität von *hostapd* um die EAP-BCA-Methode und das Blockchain-Accounting sind einige Änderungen am bestehenden Quellcode vorgenommen worden, sowie neue Quellcode-Dateien hinzugekommen.

Nachfolgend werden die gemachten Änderungen beschrieben.

Änderungen an den Makefiles: Da als Build-Management-Tool *make* zum Einsatz kommt, sind Änderungen an den *Makefiles* nötig, um die neu hinzugekommenen Quellcode-Dateien zu kompilieren.

Änderungen am Makefile *hostapd/Makefile*:

Für das Blockchain-Accounting wurde die Datei *src/ap/bc_accounting.o* zu den *Object-File-Targets* hinzugefügt:

```
OBJS += ../src/ap/bc_accounting.o
```

Für die EAP-Methode BCA wurde folgendes ergänzt:

```
ifdef CONFIG_EAP_BCA
CFLAGS += -DEAP_SERVER_BCA
OBJS += ../src/utils/cJSON.o
OBJS += ../src/eap_server/eap_server_bca.o
OBJS += ../src/eap_server/eap_server_bca_common.o
OBJS += ../src/eap_common/eap_bca_common.o
OBJS += ../src/crypto/aes-gcm.o
LIBS += -lm
NEED_AES=y
NEED_SHA256=y
endif
```

Änderungen für die EAP-BCA-Methoden Integration: Das Registrieren der EAP-BCA-Methode erfolgt in der Datei *hostapd/eap_register.c*.

Das Parsen der Konfigurationsparameter für die EAP-BCA-Methode, aus der Konfigurationsdatei *hostapd.conf*, erfolgt in der Datei *hostapd/config_file.c*. Dort werden die Parameter *eap_bca_auth_private_key*, *eap_bca_eth_auth_address*, *eap_bca_eth_auth_passphrase* und *eap_bca_eth_ipc_file_path* eingelesen (siehe 4.1.1.4 *Konfigurationsdaten*). Das Anlegen der Variablen für die Konfigurationsparameter geschieht in der Datei *src/ap/ap_config.h*, wo die Variable der Konfigurationsparameter innerhalb der Struktur `hostapd_bss_config` angelegt sind. Das wieder Freigeben von dynamischem Speicher der Variablen erfolgt in der dazugehörigen C-Quellcode-Datei *src/ap/ap_config.c* (in der Funktion `hostapd_config_free_bss`).

Die Initialisierung der allgemeinen Variablen geschieht in der Funktion *bca_global_init*, welche in der Datei *src/eap_server/eap_server_bca_common.c* beschrieben ist und in der Datei *src/ap/hostapd.c* aufgerufen wird. In der Main-Datei (*hostapd/main.c*) wird nun zusätzlich beim Deinitialisieren, die Funktion *bca_global_deinit* (aus der Datei *src/eap_server/eap_server_bca_common.c*) aufgerufen, welche die allgemeinen Variablen der EAP-BCA-Methode wieder freigibt. Diese allgemeinen Variablen der EAP-BCA-Methode sind zum Beispiel Kopien der Konfigurationsparameter, die somit innerhalb der Methoden einer EAP-Methode verfügbar gemacht werden.

Die *Vendor-Id* und der *Vendor-Type* der EAP-BCA-Methode ist in der Header-Datei *src/eap_common/eap_defs.h* definiert.

Neu für die EAP-BCA-Methode hinzugekommen sind die folgenden Quellcode-Dateien:

- *src/eap_common/eap_bca_common.c*
- *src/eap_common/eap_bca_common.h*
- *src/eap_server/eap_bca_common.h*
- *src/eap_server/eap_server_bca.c*
- *src/eap_server/eap_server_bca_common.c*

Blockchain-Accounting In der Datei *src/ap/accounting.c* werden nun die (neu hinzugekommenen) Blockchain-Accounting Start- und Stopp-Funktionen (*bc_accounting_sta_start* und *bc_accounting_sta_stop*) aufgerufen.

Diese beiden Blockchain-Accounting Funktionen sind in der neuen C-Quellcode-Datei *src/ap/bc_accounting.c* beschrieben und in der neuen C-Header-Datei *src/ap/bc_accounting.h* als Prototypen deklariert. Dort werden die Nutzungsdaten aber nur aufbereitet und in der zur Authentifizierung verwendeten EAP-BCA-Methoden-Objektinstanz abgelegt, in welcher die Nutzungsdaten so lange hinterlegt bleiben, bis es zu einer Verbindungstrennung des WLAN-Clienten kommt und die Daten dann als *Disconnect-Nachricht* im AAA-Contract hinterlegt werden.

4.1.1.3 Kompilieren

Zur Kompilierung des Quellcodes und erstellen der ausführbaren Datei von *hostapd* ist das Build-Management-Tool *make* nötig. Dieses ist standardmäßig auf dem Raspberry Pi installiert und somit kann, nach der Installation der benötigten Software-Bibliotheken, die Kompilierung bzw. Erstellung der ausführbaren Datei (`hostapd`) mit folgendem Konsolen-Befehl (innerhalb des Ordners *hostap/hostapd*) gestartet werden:

```
make clean all
```

Als externe auf dem System installierte Software-Bibliotheken werden benötigt:

ssh Stellt (Open)SSL Funktionen bereit.

dnet Stellt Ressourcen für die DECnet-Entwicklung unter Linux bereit.

nl Stellt eine Schnittstelle zum Arbeiten mit *netlink*-Verbindungen bereit.

Die Installation der benötigten Software-Bibliotheken ist auf dem Raspberry Pi (Debian) mit dem folgenden Konsolen-Befehl möglich:

```
sudo apt-get install -y libssh-dev libdnet-dev libnl-dev
```

Angemerkt sei, das in der Build-Konfigurationsdatei *.config* der Parameter `CONFIG_EAP_BCA` auf `y` gesetzt sein muss um die EAP-Methode BCA für den Build zu Aktivieren.

4.1.1.4 Konfigurationsdaten

EAP-BCA-Konfigurationsausschnitt mit Beispieldaten aus der Datei *hostapd.conf* (vollständige Datei siehe Anhang A.3 *hostapd*):

```
eap_bca_auth_private_key=0291
    ↪ e444ecb0cba525a13b490c25ab5b05d08e00590538e744628fe085e180d1
eap_bca_eth_ipc_file_path=/home/pi/.ethereum/testnet/geth.ipc
eap_bca_eth_auth_address=08aef3ccd9a2f8b73de99ace3cd35aa80f22f88f
eap_bca_eth_auth_passphrase=""
```

Erläuterung der gezeigten Konfigurationsdaten:

eap_bca_auth_private_key
: Setzt den privaten ECDSA-Schlüssel des Authenticators (32 Bytes Länge) auf den nachfolgenden hexadezimalen Wert.

eap_bca_eth_ipc_file_path
: Setzt den Pfad zur IPC-Datei des Ethereum-Clienten *geth*.

eap_bca_eth_auth_address
: Setzt die Ethereum-Adresse des Ethereum-Accounts des Authenticators (20 Bytes Länge) auf den nachfolgenden hexadezimalen Wert.

eap_bca_eth_auth_passphrase
: Setzt die *passphrase* (Entschlüsselungs-Passwort) des Ethereum-Accounts des Authenticators für *geth*.

EAP-BCA-Nutzer-Konfigurationsausschnitt mit Beispieldaten aus der Datei *hostapd.eap_user* (vollständige Datei siehe Anhang A.3 *hostapd*):

```
"testNetworkId"    BCA    55110407995624598
    ↪ c6a935c57e31aa0a04d449425f98e1db744c0630a7a6799a6ec82f1
```

Erläuterung der gezeigten Daten aus der Konfigurationsdatei *hostapd.eap_user*:

Eine Zeile der Konfigurationsdatei entspricht einer Nutzerfreigabe. Dabei werden im gezeigten Beispiel drei Parameter gesetzt.

Der erste Parameter ist er Nutzername (EAP-Identität; `identity`), der bei EAP-BCA der bcNetworkId entspricht (mehr hierzu siehe 3.1.3 *EAP-Methode: Identity*).

Der zweite Parameter `BCA` gibt an welche EAP-Methoden für den Nutzer erlaubt sind (in diesem Fall nur die EAP-BCA-Methode).

Der dritte Parameter ist das Nutzerpasswort, hier als hexadezimaler Wert angegeben, mit einer Länge von 36 Bytes. Dieser binäre Passwortwert setzt sich hier aus der Ethereum-Adresse des AAA-Contracts (mit 20 Bytes) und dem (für den jeweiligen AAA-Contract verwendeten) PSK (mit 16 Bytes) zusammen.

Nachfolgend als Pseudocode gezeigt:

```
userpassword = contractAddress ∘ contractPSK
```

In den gezeigten Beispieldaten entspricht somit:

```
contractAddress = 55110407995624598c6a935c57e31aa0a04d4494
contractPSK     = 25f98e1db744c0630a7a6799a6ec82f1
```

4.1.1.5 Nutzung von hostapd

Um *hostapd* mit der EAP-BCA-Methode zu nutzen, muss der Ethereum-Client auf dem System laufen. Der Ethereum-Client lässt sich mit folgendem Konsolen-Befehl starten:

```
geth --fast
```

Das Starten des Ethereum-Clients im *Testnet* (der Test-Blockchain für Entwicklungszwecke) ist mit folgendem Konsolen-Befehl möglich:

```
geth --testnet --fast
```

Das Ausführen (Starten) von *hostapd* mit der Beispiel-Konfiguration ist mit folgendem Konsolen-Befehl (innerhalb des Ordners *hostap/hostapd*) möglich:

```
sudo ./hostapd hostapd.conf
```

Detaillierte Debugging-Nachrichten sind mit dem Zusatz `-dd` anzeigbar:

```
sudo ./hostapd -dd hostapd.conf
```

Im Vorfeld sollte sichergestellt werden, dass die angegebene WLAN-Schnittstelle, hier *wlan0*, nicht schon von der Installierten und im Hintergrund ausgeführten Version von *wpa_supplicant* belegt ist. Diese freizugeben bzw. *wpa_supplicant* zu schließen, ist mit folgendem Konsolen-Befehl möglich:

```
sudo pkill wpa_supplicant
```

4.1.2 wpa_supplicant

Die Software *wpa_supplicant* ist ein so genanter *WPA-Supplicant* (Komponente die bei IEEE 802.1X/WPA in WLAN-Clients eingesetzt wird). Wie *hostapd* ist auch *wpa_supplicant*

ein User-Space-Daemon und unterstützt WPA und WPA2 sowie alle gängigen EAP-Methoden [4].

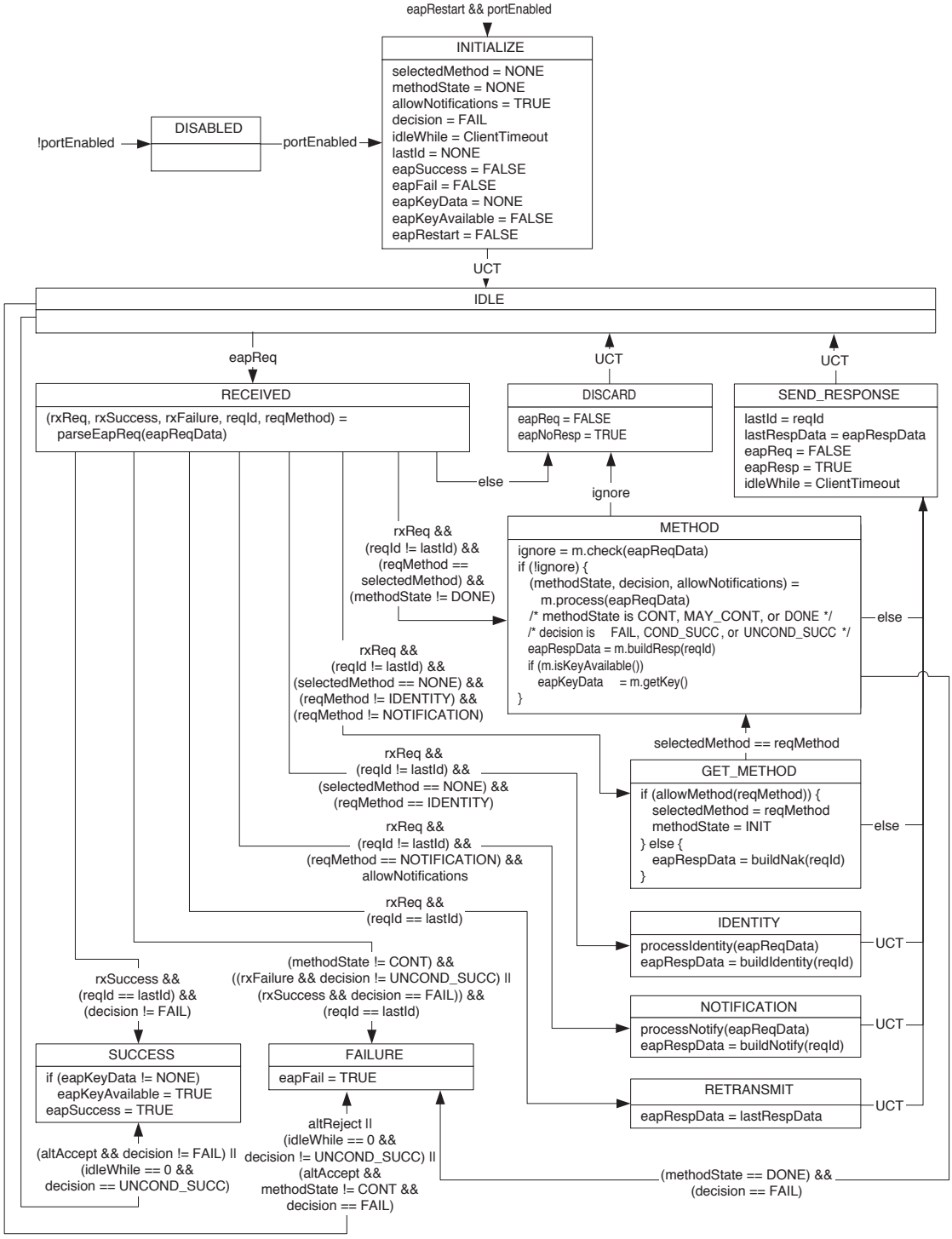


Abbildung 4.2: *EAP Peer State-Machine [36]*

In Abbildung 4.2 ist ersichtlich, dass jede EAP-Methode im Peer eine Schnittstelle aus Methoden zur Verfügung stellen muss.

Die Schnittstellen-Methoden einer EAP-Methode (laut RFC 4137 State-Machine) eines EAP-Peers sind:

`check(eapReqData)`
: Überprüft den eingegangenen EAP-Request und entscheidet ob dieser ignoriert werden soll.

`process(eapReqData)`
: Verarbeitet den eingegangenen EAP-Request.

`buildResp(reqId)`
: Erzeugt den aktuellen EAP-Response der EAP-Methode.

`methodState()`
: Gibt den aktuellen Status der EAP-Methode zurück (*INIT*, *CONT*, *MAY_CONT*, *DONE*).

`decision()`
: Gibt die Authentifizierungsentscheidung der EAP-Methode zurück (*FAIL*, *COND_SUCC*, *UNCOND_SUCC*).

`getKey()`
: Gibt das ausgehandelte Schlüsselmaterial zurück.

Diese Schnittstelle ist für die EAP-BCA-Methode des Authenticators in der Quellcode-Datei *src/eap_server/eap_server_bca.c* implementiert.

4.1.2.1 Kompilieren

Zur Kompilierung des Quellcodes und erstellen der ausführbaren Datei von *wpa_supplicant* ist das Build-Management-Tool *make* nötig. Dieses ist standardmäßig auf dem Raspberry Pi installiert und somit kann, nach der Installation der benötigten Software-Bibliotheken, die Kompilierung bzw. Erstellung der ausführbaren Datei (`wpa_supplicant`) mit folgendem Konsolen-Befehl (innerhalb des Ordners *hostap/wpa_supplicant*) gestartet werden:

```
make clean all
```

Als externe auf dem System installierte Software-Bibliotheken werden benötigt:

ssh Stellt (Open)SSL Funktionen bereit.

dnet Stellt Ressourcen für die DECnet-Entwicklung unter Linux bereit.

nl Stellt eine Schnittstelle zum Arbeiten mit *netlink*-Verbindungen bereit.

Die Installation der benötigten Software-Bibliotheken ist auf dem Raspberry Pi (Debian) mit dem folgenden Konsolen-Befehl möglich:

```
sudo apt-get install -y libssh-dev libdnet-dev libnl-dev
```

4.1.2.2 Konfigurationsdaten

EAP-BCA-Konfigurationsausschnitt mit Beispieldaten aus der Datei *wpa_supplicant.conf* (vollständige Datei siehe Anhang A.4 *wpa_supplicant*):

```
network={
    ssid="bcatestap"
    key_mgmt=WPA-EAP
    eap=BCA
    identity="testNetworkId"
    password=7b4b33b430a80f4f05bab3dde8ff3efb0357adc730c8170ca548dbafb5bbb2b7
        ↪ f3ad03543fb292608aa39dda5f44548e9c4c3390c6b2f840cfb93c221e9f22f7
}
```

Erläuterung relevanter Daten aus der Konfigurationsdatei *wpa_supplicant.conf*:

Der Wert des Parameters `identity` ist den Nutzername (EAP-Identität) und entspricht bei EAP-BCA der *bcNetworkId* mit der sich der Client verbinden möchte (mehr hierzu siehe 3.1.3 *EAP-Methode: Identity*).

Das Nutzerpasswort (`password`) ist ein 64 Bytes langer hexadezimale angegebener Wert, der sich aus dem privaten ECDSA-Schlüssel des Peers (mit 32 Bytes Länge; `peerPrivateKey`) und dem Hashwert des öffentlichen ECDSA-Schlüssels der CA (mit 32 Bytes Länge; `caKeyHash`) des jeweiligen zu verwendeten Netzes bzw. des AAA-Contract der dieses Netz verwaltet, zusammensetzt.

Nachfolgend als Pseudocode gezeigt:

```
password = peerPrivateKey ∘ caKeyHash
```

In der Beispiel-Konfiguration ist somit entsprechend:

```
peerPrivateKey = 7b4b33b430a80f4f05bab3dde8ff3efb0357adc730c8170ca548dbafb5bbb2b7
caKeyHash      = f3ad03543fb292608aa39dda5f44548e9c4c3390c6b2f840cfb93c221e9f22f7
```

4.1.2.3 Nutzung von wpa_supplicant

Das Ausführen (Starten) von *wpa_supplicant* mit der Beispiel-Konfiguration, ist mit folgendem Konsolen-Befehl (innerhalb des Ordners *hostap/wpa_supplicant*) möglich:

```
sudo ./wpa_supplicant -iwlan0 -c wpa_supplicant.conf
```

Detaillierte Debugging-Nachrichten sind mit dem Zusatz `-dd` anzeigbar:

```
sudo ./wpa_supplicant -dd -iwlan0 -c wpa_supplicant.conf
```

Im Vorfeld sollte sichergegangen werden das die angegebene WLAN-Schnittstelle, hier *wlan0*, nicht schon von der Installierten und im Hintergrund offenen Version von *wpa_supplicant* belegt ist. Um diese Freizugeben bzw. *wpa_supplicant* zu schliessen, ist mit folgendem Konsolen-Befehl möglich:

```
sudo pkill wpa_supplicant
```

4.2 AAA-Contract

4.2.1 Storage-Variablen

```
address  owner
bytes    caPublicKey

mapping(bytes32 => AccessEntry) accessList
mapping(address => AdminEntry)  adminList

mapping(bytes32 => bytes32)     authKeySignPart0
mapping(bytes32 => bytes32)     authKeySignPart1
mapping(address => bytes32)     authAddressIdx

mapping(bytes32 => bytes32)     accounting
```

Im Unterschied zum Konzept des AAA-Contracts ist es in der Implementierung des AAA-Contracts von Vorteil, für die Schlüssel-Signatur des Authenticators (`authKeySign`), statt eines Mappings auf ein Byte-Array, zwei getrennte Mappings mit Variablen vom Typ `bytes32` zu verwenden.

Dies hat den Hintergrund, dass in Ethereum für das Speichern eines Key-Value-Paares im Bytecode immer der (Ethereum-Standard-) Typ `bytes32` für Key und Value verwendet wird und ein variables Array mit einem Header belastet ist, der die Länge des Arrays speichert.

Diese Längenangabe ist hier aber unnötig, da die Signatur bei der festgelegten EC-Kurve auch immer eine feste Größe besitzt.

Somit kann durch das "manuelle" Aufteilen der Daten der Signatur hier (unnötiger) Speicherplatz in der Blockchain und somit auch Kosten (in Form von *Gas*) bei dem Hinzufügen eines Authenticators zum AAA-Backend-Contract eingespart werden.

In diesem Fall kann eine von drei Speicheroperationen eingespart werden, was einer Einsparung von 33% oder 20000 *Gas* entspricht.

Des Weiteren währe dies beim öffentlichen Schlüssel der CA (`caPublicKey`) auch möglich gewesen. Es wurde hier aber darauf verzichtet, da die Einsparungen hier nur einmalig bei Erstellen des Contract oder womöglich beim Ändern des Schlüssels auftreten und somit in keinem Verhältnis zu einer gewünschten guten Lesbarkeit des Contract-Programmcodes stehen.

4.2.2 Methoden-Kosten

Die Beschreibungen zu den einzelnen Methoden finden sich im Konzept des AAA-Contracts, siehe 3.2.1 *AAA-Contract*.

Nachfolgend werden ergänzend die Kosten (in *Gas*) für die in der Blockchain auszuführenden Methoden und die Erstellung bzw. Terminierung des Contracts angegeben.

Die Angaben sind dabei mit dem Contract-Bytecode der mit dem Contract-Programmcode der unter 4.2.3 angegebene ist, ermittelt worden.

Lokal-auszuführende Methoden werden nicht angegeben, da diese keine Gas-Kosten zur folge haben.

4.2.2.1 Initialisierung des Contracts

AAA(bytes caPubKey) 2123974 *Gas*

Die *Gas*-Kosten für die Erstellung eines AAA-Contracts mit dem gezeigten Contract-Programmcode (siehe 4.2.3) kommen auf 2123974 *Gas*.

4.2.2.2 Contract-Verwaltungs-Methoden

setOwner(address newOwner) 28848 *Gas*

destruct() 13510 *Gas*

setCAPublicKey(bytes caPubKey) 42596 *Gas*

4.2.2.3 Administrator-Management

addAdmin(address adminAddress) 43820 *Gas*

removeAdmin(address adminAddress) 14315 *Gas*

4.2.2.4 User-Management

addAccess(bytes32 accessToken) 44648 *Gas*

removeAccess(bytes32 accessToken) 14740 *Gas*

4.2.2.5 Authenticator-Management

addAuthenticator(...) 91040 *Gas*

removeAuthenticator(address authAddress) 19570 *Gas*

4.2.2.6 Accounting

updateAccounting(bytes32 accessToken, bytes32 status) 60907 *Gas* (erstes Accounting eines Nutzers: 75907 *Gas*)

4.2.3 Contract-Programmcode

Contract-Programmcode in der Programmiersprache Solidity:

```
pragma solidity ^0.4.9;

/// @title Block chain network AAA smart contract
contract AAA {

    struct AccessEntry {
        bool hasAccess;
    }

    struct AdminEntry {
        bool isAdmin;
    }


    address owner;
    bytes   caPublicKey;
    mapping(bytes32 => AccessEntry) accessList;
    mapping(address => AdminEntry)  adminList;

    // mapping to authenticator key signatures
    // key = authKeyHash
    // value splitet in two parts of the authenticator key signature
    mapping(bytes32 => bytes32)     authKeySignPart0;
    mapping(bytes32 => bytes32)     authKeySignPart1;

    // mapping to index the authenticator ethereum address
    // key = authenticator ethereum address
    // value = authKeyHash
    mapping(address => bytes32)     authAddressIdx;

    // key = sha3(accessToken + idx)
    // value = status array
    // idx = 0 -> entry count
    mapping(bytes32 => bytes32)     accounting;


    function AAA(bytes caPubKey) {
        adminList[msg.sender].isAdmin = true;
        owner = msg.sender;
        caPublicKey = caPubKey;
    }


    // smart contract ownership methods

    function setOwner(address newOwner) {
        // only the current owner can set the new owner
        require(msg.sender == owner);
        // check the new owner balance
        require(newOwner.balance > 0);

        owner = newOwner;
    }

    function destruct() {
        require(msg.sender == owner);

        selfdestruct(owner);
    }


    function setCAPublicKey(bytes caPubKey) {
        require(msg.sender == owner);

        caPublicKey = caPubKey;
```

```
}

function getCAPublicKey() constant
        returns (bytes)
{
    return caPublicKey;
}

// Admin management methods

function addAdmin(address adminAddress) {
    require(adminList[msg.sender].isAdmin == true ||
            msg.sender == owner);
    adminList[adminAddress].isAdmin = true;
}

function isAdmin(address adminAddress) constant
        returns (bool isadmin)
{
    isadmin = !(adminList[adminAddress].isAdmin != true);
}

function removeAdmin(address adminAddress) {
    require(adminList[msg.sender].isAdmin == true ||
            msg.sender == owner);
    delete adminList[adminAddress];
}


// static local call methods

function accessToken(bytes32 keyhash, bytes psk) constant
        returns (bytes32 token)
{
    bytes memory c = new bytes(32 + psk.length);
    uint cp = 0;
    for (uint i = 0; i < 32; i++) {
        c[cp++] = keyhash[i];
    }
    for (i = 0; i < psk.length; i++) {
        c[cp++] = psk[i];
    }
    token = sha256(c);
}


// access methods

function addAccess(bytes32 accessToken) {
    require(adminList[msg.sender].isAdmin == true);
    accessList[accessToken].hasAccess = true;
}

function hasAccess(bytes32 accessToken) constant
        returns (bool bHasAccess)
{
    bHasAccess = false;

    AccessEntry ae = accessList[accessToken];
    if (ae.hasAccess == true) {
        bHasAccess = true;
    }
}

function removeAccess(bytes32 accessToken) {
```

```
        require(adminList[msg.sender].isAdmin == true);

        delete accessList[accessToken];
    }


    // Authenticator management methods

    function addAuthenticator(address authAddress,
            bytes32 authKeyHash,
            bytes keySign)
    {
        require(adminList[msg.sender].isAdmin == true);
        bytes32 t0;
        bytes32 t1;

        assembly {
            t0 := mload(add(keySign, 32))
            t1 := mload(add(keySign, 64))
        }

        authKeySignPart0[authKeyHash] = t0;
        authKeySignPart1[authKeyHash] = t1;
        authAddressIdx[authAddress] = authKeyHash;
    }

    function getAuthKeySign(bytes32 authKeyHash) constant
            returns (bool isFound, bytes sign)
    {
        if (authKeySignPart0[authKeyHash] == 0) {
            isFound = false;
            return;
        }

        sign = new bytes(64);
        for (uint i = 0; i < 32; i++) {
            sign[i] = authKeySignPart0[authKeyHash][i];
        }
        for (i = 0; i < 32; i++) {
            sign[i + 32] = authKeySignPart1[authKeyHash][i];
        }

        isFound = true;
    }

    function removeAuthenticator(address authAddress) {
        require(adminList[msg.sender].isAdmin == true);

        bytes32 idx = authAddressIdx[authAddress];
        delete authKeySignPart0[idx];
        delete authKeySignPart1[idx];
        delete authAddressIdx[authAddress];
    }


    // Accounting methods

    function updateAccounting(bytes32 accessToken, bytes32 status) {
        require(authAddressIdx[msg.sender] != 0 ||
                adminList[msg.sender].isAdmin == true);

        //bytes32 idx = 0;
        bytes memory keybase = new bytes(32 + 8);
        for (uint i = 0; i < 32; i++) {
            keybase[i] = accessToken[i];
```

```
        }
        for (i = 0; i < 8; i++) {
            keybase[i + 32] = 0x0;  // = idx[i];
        }
        bytes32 countKey = sha3(keybase);
        uint64 entryCount = uint64(accounting[countKey]);

        uint64 idx = entryCount + 1;
        for (i = 0; i < 8; i++) {
            keybase[i + 32] = bytes8(idx)[i];
        }
        bytes32 idxKey = sha3(keybase);

        entryCount = entryCount + 1;
        accounting[countKey] = bytes32(entryCount);
        accounting[idxKey] = status;
    }

    function getAccounting(bytes32 accessToken, uint64 index) constant
            returns (bytes32)
    {
        require(adminList[msg.sender].isAdmin == true ||
                authAddressIdx[msg.sender] != 0);

        uint64 idx = index + 1;
        bytes memory keybase = new bytes(32 + 8);
        for (uint i = 0; i < 32; i++) {
            keybase[i] = accessToken[i];
        }
        for (i = 0; i < 8; i++) {
            keybase[i + 32] = bytes8(idx)[i];
        }
        bytes32 idxKey = sha3(keybase);
        return accounting[idxKey];
    }

    function getAccountingEntryCount(bytes32 accessToken) constant
            returns (uint64)
    {
        require(adminList[msg.sender].isAdmin == true ||
                authAddressIdx[msg.sender] != 0);

        //bytes32 idx = 0;
        bytes memory keybase = new bytes(32 + 8);
        for (uint i = 0; i < 32; i++) {
            keybase[i] = accessToken[i];
        }
        for (i = 0; i < 8; i++) {
            keybase[i + 32] = 0x0;  // = idx[i];
        }
        bytes32 countKey = sha3(keybase);
        return uint64(accounting[countKey]);
    }

}
```

Kapitel 5

Ergebnisse / Resultate

Das Ergebnis dieser Arbeit ist ein voll funktionsfähiges Blockchain-basiertes Triple-A-System.

Um die Performanz dieses Systems mit dem anderer Triple-A-Systeme vergleichen zu können, werden nachfolgend die Ergebnisse eines Verbindungsaufbau-Testes gezeigt und ausgewertet.

Als repräsentativen Kandidaten und Gegenspieler zu BCA wird EAP-TLS, mit lokalem Authentifizierungs-Server, eingesetzt.

Der lokale Authentifizierungs-Server soll die normalerweise extrem-schnelle Backend-Authentifizierungs- und Autorisierungs-Infrastruktur eines normalen Triple-A-Systems abbilden, da die Antwortzeiten eines solchen Systems bei der Verzögerung einer (schwachen) Access-Point-Hasrdware vernachlässigbar sind.

Somit hat die Authentifizierung mit EAP-TLS hier einen zeitlichen Vorteil gegenüber EAP-BCA, da bei BCA zwar der Blockchain-Client auch lokal (auf dem gleichen Gerät läuft) aber nicht in die gleiche Applikation integriert ist. Somit muss zum einen noch mit dem Blockchain-Clienten kommunizieren werden und zum Anderen führt die Ausführung des (nicht für Embedded-Geräte optimierten) Blockchain-Clienten zu einer Belastung des Systems, die die Authentifizierung ausbremst.

EAP-TLS ist dabei besonders repräsentativ, da diese EAP-Methode TLS direkt zur Authentifizierung verwendet, was bedeutet, dass die Authentifizierung damit immer schneller als bei EAP-Methoden wie TTLS oder PEAP ist. Diese nutzen TLS nur zum Tunneln der Daten einer eigentlichen EAP-Authentifizierungs-Methode (in einer zusätzlichen EAP-Schicht) und stellen somit nur eine von zwei Phasen der EAP-Authentifizierung dar. Somit kann eine EAP-Authentifizierung mit TTLS oder PEAP nicht schneller sein, als eine mit EAP-TLS.

Das Sicherheitsniveau von EAP-TLS ist grundsätzlich abhängig von der Stärke der, in der Zertifikaten verwendeten, Schlüssel (und gegebenenfalls von den DH-Schlüssel).

Da die unterschiedlichen TLS-Implementierungen im Allgemeinen noch massive Probleme mit EC-Schlüsseln in Zertifikaten haben, werden hier (wie im Allgemeinen zu fast 100% auch) RSA-Schlüssel verwendet.

Für die Tests werden, in den Zertifikaten, öffentliche RSA-Schlüssel mit einer Schlüssellänge von 3072 Bit verwendet. Der verwendete Signatur-Algorithmus ist *sha256WithRSAEncryption*. Was insgesamt dem Sicherheitsniveau von EAP-BCA (in dieser Arbeit von 128 Bit) entspricht. Aber im Gegensatz zu EAP-BCA verfügt die EAP-TLS Authentifizierung nicht über einen Identitätsschutz, dieser zusätzliche Schutz

der Identität des Nutzers müsste bei EAP-TLS durch einen zusätzlichen TTLS oder PEAP Tunnel (in einer ersten EAP-Phase) geschützt werden.

5.1 Verbindungsaufbau-Test

Um die Performanz der EAP-BCA-Methode abzuschätzen bzw. einen Vergleich mit anderen EAP-Methoden herstellen zu können, sind nachfolgend die Zeiten von Verbindungsaufbauen gezeigt.

Die zeitlichen Werte wurden dabei aus Debug-Log-Nachrichten (mit Zeitangaben auf eine Mikrosekunde exakt; aus der Konsolen-Ausgabe), über den nachfolgend gezeigten Konsolen-Befehl, ermittelt:

```
sudo wpa_supplicant -d -t -iwlan0 -c wpa_supplicant.conf
```

Die Zeit für den kompletten Verbindungsaufbau (Verbindungszeit) ist dabei von der Log-Mitteilung mit den Versions-Informationen bis zu der Log-Mitteilung "`EAPOL authentication completed - result=SUCCESS`" festgelegt.

Die durchschnittliche Verbindungszeit beträgt bei EAP-TLS 2,57s und bei EAP-BCA 1,13s.

Die reine Ausführungszeit der EAP-Methode beträgt bei EAP-TLS 1,514625s und bei EAP-BCA 0,117197s.

Dabei wurde bei EAP-TLS die Ausführungszeit von der Log-Mitteilung "`EAP-TLS: Start`" bis zu der Log-Mitteilung "`EAP: Received EAP-Success`" gemessen und bei EAP-BCA die Ausführungszeit von der Log-Mitteilung "`EAP-BCA: start process ...`" bis zu der Log-Mitteilung "`EAP: Received EAP-Success`" gemessen.

	EAP-TLS		EAP-BCA
Verbindungszeit	2,57s	$\xrightarrow{\text{mehr als doppelt so schnell}}$	1,13s
Ausführungszeit	1,514625s	$\xrightarrow{\text{12,9 mal schneller}}$	0,117197s

Die Ausführungszeit der EAP-BCA-Methode ist im Vergleich mit der EAP-TLS-Methode fast 13 mal so schnell, was im Endeffekt die Verbindungsgeschwindigkeit mehr als verdoppelt.

Die sehr kurze Ausführungszeit der EAP-BCA-Methode lässt sich dadurch erklären, dass bei BCA nur ein EAP-Request mit einem damit verbundenen EAP-Response ausgetauscht werden muss. Während bei EAP-TLS die Anzahl der EAP-Requests/-Responses um ein vielfaches höher ist. In diesem Test z.B. waren für die EAP-TLS-Authentifizierung 7 EAP-Request und EAP-Responses nötig.

Kapitel 6

Zusammenfassung

Das primäre Ziel der Arbeit die Konzeption eines Blockchain-basierten Triple-A-Systems (AAA-Systems) und die anschließende Implementierung einer EAP-Methode (EAP-Erweiterung) wurde voll erfüllt.

In einer Voruntersuchung wurde dazu untersucht, welche Blockchain, Implementierungsumgebung und Kryptosysteme infrage kommen. Die Wahl viel hier auf die Blockchain Ethereum, als optimale Implementierungsumgebung hat sich ein *Raspberry Pi 3* System ergeben und bei den Kryptosystemen viel die Wahl auf Kryptosysteme, die der TLS Cipher Suite `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` entsprechen.

In der Konzeptionsphase entstand ein Authentifizierungsprotokoll, das mit EAP verwendet werden kann bzw. eine neue EAP-Methode (EAP-BCA). Des Weiteren wurde der grundlegende Aufbau des Blockchain-basierten Triple-A-System-Backends konzipiert, sowie das Nutzungskonzept beschrieben.

Die Implementierung erfolgte als Erweiterung der WLAN-Access-Point-Software *hostapd* und der WLAN-Clienten-Software *wpa_supplicant*.

Die Ergebnisse eines Verbindungsgeschwindigkeits-Tests sind dabei, im Vergleich mit anderen EAP-Methoden (mit vergleichbarem Sicherheitsniveau) erstaunlich gut. Was zeigt das ein Blockchain-basiertes Triple-A-System, mit einem dementsprechend modernen Authentifizierungsprotokoll (EAP-BCA-Protokoll), nicht nur praktisch machbar ist, sondern auch sehr performant sein kann.

Das sekundäre Ziel, die Entwicklung eines Ansatzes zur automatischen Verrechnung (Payment) im Blockchain-basierten Triple-A-System, konnte aufgrund der eingeschränkten zeitlichen Vorgaben der Masterarbeit leider nicht mehr vollständig bearbeitet werden.

Einzig das grundlegende Konzept eines Micropayment-Contracts zur Abrechnung (siehe 2.3 *Micropayment*) und ein leicht erweiterbarer AAA-Contract konnten vorbereitet werden.

Kapitel 7

Ausblick

Diese Arbeit zeigt, dass die Blockchain-Technologie auch im Bereich der Autorisierung und dem Accounting großes Potenzial besitzt. In naher Zukunft besteht somit die Möglichkeit auf sich selbstverwaltende Triple-A-Systeme bzw. Netzwerk-/Internet-Anbieter in Form von Smart-Contracts in der Blockchain, die vollautomatisch und autonom sich um die Verwaltung und Abrechnung von Nutzern kümmern.

Auch besteht die Möglichkeit, die Autorisierung über die Grenze von Netzwerken hinaus zu erweitern. Somit wäre es möglich Zugriffskontrollen für Räume, Einrichtungen, Gelände (z. B. für Hotelzimmer, Parks, usw.), Fahrzeuge (z. B. bei Carsharing und Fahrradverleihsysteme) und Geräte bzw. Maschinen (Schlagworte sind hier allgemein Sharing und Industrie 4.0) in einem Blockchain-Contract zu verwalten und die Nutzung abzurechnen.

Literaturverzeichnis

[1] hostapd. http://w1.fi/hostapd/. (Seite 64)

[2] Hyperledger. https://www.hyperledger.org. (Seiten 15 und 16)

[3] Namecoin. https://www.namecoin.org. (Seite 15)

[4] wpa_supplicant. http://w1.fi/wpa_supplicant/. (Seite 70)

[5] Etherscan - chart - blocktime. https://etherscan.io/chart/blocktime, 2017. (Seite 7)

[6] Litecoin. https://litecoin.org/, 2017. (Seite 7)

[7] Nxt blockchain explorer. https://nxtportal.org/charts/, 2017. (Seite 7)

[8] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. Extensible authentication protocol (eap). RFC 3748 (Proposed Standard), June 2004. Updated by RFCs 5247, 7057. (Seiten ii, iii, 9, 10, 11, 12, 13, 33 und 90)

[9] bitcoin.org. bitcoin.org developer guide. https://bitcoin.org/en/developer-guide#contracts, 2017. (Seite 7)

[10] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027, 7919. (Seiten 22 und 23)

[11] P. Chown. Advanced encryption standard (aes) ciphersuites for transport layer security (tls). RFC 3268 (Proposed Standard), June 2002. Obsoleted by RFC 5246. (Seite 28)

[12] P. Congdon, B. Aboba, A. Smith, G. Zorn, and J. Roese. Ieee 802.1x remote authentication dial in user service (radius) usage guidelines. RFC 3580 (Informational), September 2003. Updated by RFC 7268. (Seite ii)

[13] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. (Seiten 30 und 33)

[14] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. https://www-ee.stanford.edu/~hellman/publications/24.pdf, November 1976. (Seite 21)

[15] Information Technology DIN-Normenausschuss Informationstechnik und Anwendungen (NIA) and selected IT Applications Standards Committee. Informationstechnik - it-sicherheitsverfahren - informationssicherheits-managementsysteme - überblick und terminologie (iso/iec 27000:2009), 7 2011. (Seite 8)

[16] V. Fajardo, J. Arkko, J. Loughney, and G. Zorn. Diameter Base Protocol. RFC 6733 (Proposed Standard), October 2012. Updated by RFC 7075. (Seite 9)

[17] S. Frankel, R. Glenn, and S. Kelly. The aes-cbc cipher algorithm and its use with ipsec. RFC 3602 (Proposed Standard), September 2003. (Seite 28)

[18] Mike Hearn and Jeremy Spilman. Bitcoin contracts. `https://en.bitcoin.it/w/index.php?title=Contract&oldid=63826`, August 2017. [Online; Stand 10. August 2017]. (Seite 17)

[19] K. Igoe and J. Solinas. Aes galois counter mode for the secure shell transport layer protocol. RFC 5647 (Informational), August 2009. (Seite 28)

[20] Nicholas Jansma and Brandon Arrendondo. Performance comparison of elliptic curve and rsa digital signatures. `http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf`, apr 2004. (Seite 22)

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151. (Seite 30)

[22] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, T. Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: A scalable blockchain database. ascribe GmbH, Berlin, Germany, jun 2016. (Seite 15)

[23] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard), January 2008. (Seite 28)

[24] D. McGrew and K. Igoe. Aes-gcm authenticated encryption in the secure real-time transport protocol (srtp). RFC 7714 (Proposed Standard), December 2015. (Seite 28)

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. (Seiten iii, 1, 2, 7 und 16)

[26] National Institute of Standards and Technology. Advanced encryption standard (aes). FIPS PUB 197, November 2001. (Seite 28)

[27] National Institute of Standards and Technology. Digital signature standard (dss). National Intitute of Standards and Technology, FIPS PUB 186-4, July 2013. (Seiten 21 und 22)

[28] National Institute of Standards and Technology. Recommendation for key management, part 1: General. National Intitute of Standards and Technology, Special Publication 800-57 Part 1 Revision 4, July 2013. (Seiten 28, 30 und 40)

[29] J. Peterson. S/mime advanced encryption standard (aes) requirement for the session initiation protocol (sip). RFC 3853 (Proposed Standard), July 2004. (Seite 28)

[30] K. Raeburn. Advanced encryption standard (aes) encryption for kerberos 5. RFC 3962 (Proposed Standard), February 2005. (Seite 28)

[31] E. Rescorla. Tls elliptic curve cipher suites with sha-256/384 and aes galois counter mode (gcm). RFC 5289 (Informational), August 2008. (Seiten 29 und 30)

[32] Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Standard 1.0, Certicom Corp., September 2000. (Seiten 23, 24, 87, 88, 89 und 90)

[33] J. Salowey, A. Choudhury, and D. McGrew. Aes galois counter mode (gcm) cipher suites for tls. RFC 5288 (Proposed Standard), August 2008. (Seite 29)

[34] IEEE Computer Society. *IEEE Std 802.1X-2004 (Revision of IEEE Std 802.1X-2001): IEEE Standard for Local and metropolitan area networks - Port-Based Network Access Control*. IEEE, December 2004. (Seiten iii, 13 und 14)

[35] IEEE Computer Society. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE, March 2012. (Seiten iii und 14)

[36] J. Vollbrecht, P. Eronen, N. Petroni, and Y. Ohba. State machines for extensible authentication protocol (eap) peer and authenticator. RFC 4137 (Informational), August 2005. (Seiten iii, 64 und 70)

[37] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, EIP-150 REVISION, 2014. (Seiten ii, 7, 8, 16 und 17)

Index

Anhang A

Anhang

A.1 Elliptische Kurven

SEC 2: Recommended Elliptic Curve Domain Parameters[32]

Parameters	Section	Strength	Size	RSA/DSA	Koblitz / Random
secp112r1	2.2.1	56	112	512	r
secp112r2	2.2.2	56	112	512	r
secp128r1	2.3.1	64	128	704	r
secp128r2	2.3.2	64	128	704	r
secp160k1	2.4.1	80	160	1024	k
secp160r1	2.4.2	80	160	1024	r
secp160r2	2.4.3	80	160	1024	r
secp192k1	2.5.1	96	192	1536	k
secp192r1	2.5.2	96	192	1536	r
secp224k1	2.6.1	112	224	2048	k
secp224r1	2.6.2	112	224	2048	r
secp256k1	2.7.1	128	256	3072	k
secp256r1	2.7.2	128	256	3072	r
secp384r1	2.8.1	192	384	7680	r
secp521r1	2.9.1	256	521	15360	r

Tabelle A.1: *SEC 2: Recommended Elliptic Curve Domain Parameters[32] Table 1: Properties of Recommended Elliptic Curve Domain Parameters over* $\mathbb{F}_p$

Parameters	Section	ANSI X9.62	ANSI X9.63	echeck	IEEE P1363	IPSec	NIST	WAP
secp112r1	2.2.1	-	-	-	c	c	-	r
secp112r2	2.2.2	-	-	-	c	c	-	c
secp128r1	2.3.1	-	-	-	c	c	-	c
secp128r2	2.3.2	-	-	-	c	c	-	c
secp160k1	2.4.1	c	r	c	c	c	-	c
secp160r1	2.4.2	c	c	c	c	c	-	r
secp160r2	2.4.3	c	r	c	c	c	-	c
secp192k1	2.5.1	c	r	c	c	c	-	c
secp192r1	2.5.2	r	r	c	c	c	r	c
secp224k1	2.6.1	c	r	c	c	c	-	c
secp224r1	2.6.2	c	r	c	c	c	r	c
secp256k1	2.7.1	c	r	c	c	c	-	c
secp256r1	2.7.2	r	r	c	c	c	r	c
secp384r1	2.8.1	c	r	c	c	c	r	c
secp521r1	2.9.1	c	r	c	c	c	r	c

Tabelle A.2: *SEC 2: Recommended Elliptic Curve Domain Parameters[32] Table 2: Status of Recommended Elliptic Curve Domain Parameters over* $\mathbb{F}_p$

Parameters	Section	Strength	Size	RSA/DSA	Koblitz / Random
sect113r1	3.2.1	56	113	512	r
sect113r2	3.2.2	56	113	512	r
sect131r1	3.3.1	64	131	704	r
sect131r2	3.3.2	64	131	704	r
sect163k1	3.4.1	80	163	1024	k
sect163r1	3.4.2	80	163	1024	r
sect163r2	3.4.3	80	163	1024	r
sect193r1	3.5.1	96	193	1536	r
sect193r2	3.5.2	96	193	1536	r
sect233k1	3.6.1	112	233	2240	k
sect233r1	3.6.2	112	233	2240	r
sect239k1	3.7.1	115	239	2304	k
sect283k1	3.8.1	128	283	3456	k
sect283r1	3.8.2	128	283	3456	r
sect409k1	3.9.1	192	409	7680	k
sect409r1	3.9.2	192	409	7680	r
sect571k1	3.10.1	256	571	15360	k
sect571r1	3.10.2	256	571	15360	r

Tabelle A.3: *SEC 2: Recommended Elliptic Curve Domain Parameters[32] Table 4: Properties of Recommended Elliptic Curve Domain Parameters over* $\mathbb{F}_{2^m}$

Parameters	Section	ANSI X9.62	ANSI X9.63	echeck	IEEE P1363	IPSec	NIST	WAP
sect113r1	3.2.1	-	-	-	c	c	-	r
sect113r2	3.2.2	-	-	-	c	c	-	c
sect131r1	3.3.1	-	-	-	c	c	-	c
sect131r2	3.3.2	-	-	-	c	c	-	c
sect163k1	3.4.1	c	r	r	c	r	r	r
sect163r1	3.4.2	c	c	r	c	r	-	c
sect163r2	3.4.3	c	r	r	c	c	r	c
sect193r1	3.5.1	c	r	c	c	c	-	c
sect193r2	3.5.2	c	r	c	c	c	-	c
sect233k1	3.6.1	c	r	c	c	c	r	c
sect233r1	3.6.2	c	r	c	c	c	r	c
sect239k1	3.7.1	c	c	c	c	c	-	c
sect283k1	3.8.1	c	r	r	c	r	r	c
sect283r1	3.8.2	c	r	r	c	r	r	c
sect409k1	3.9.1	c	r	c	c	c	r	c
sect409r1	3.9.2	c	r	c	c	c	r	c
sect571k1	3.10.1	c	r	c	c	c	r	c
sect571r1	3.10.2	c	r	c	c	c	r	c

Tabelle A.4: *SEC 2: Recommended Elliptic Curve Domain Parameters[32] Table 5: Status of Recommended Elliptic Curve Domain Parameters over* $\mathbb{F}_{2^m}$

A.2 EAP

Auszug aus dem IETF-Standard RFC3748 *Extensible Authentication Protocol (EAP)*, Seiten 43 und 44 [8]:

```
7.2.  Security Claims

   In order to clearly articulate the security provided by an EAP
   method, EAP method specifications MUST include a Security Claims
   section, including the following declarations:

   [a] Mechanism.  This is a statement of the authentication technology:
       certificates, pre-shared keys, passwords, token cards, etc.

   [b] Security claims.  This is a statement of the claimed security
       properties of the method, using terms defined in Section 7.2.1:
       mutual authentication, integrity protection, replay protection,
       confidentiality, key derivation, dictionary attack resistance,
       fast reconnect, cryptographic binding.  The Security Claims
       section of an EAP method specification SHOULD provide
       justification for the claims that are made.  This can be
       accomplished by including a proof in an Appendix, or including a
       reference to a proof.
```

[c] Key strength. If the method derives keys, then the effective key strength MUST be estimated. This estimate is meant for potential users of the method to determine if the keys produced are strong enough for the intended application.
The effective key strength SHOULD be stated as a number of bits, defined as follows: If the effective key strength is N bits, the best currently known methods to recover the key (with non-negligible probability) require, on average, an effort comparable to 2^(N-1) operations of a typical block cipher. The statement SHOULD be accompanied by a short rationale, explaining how this number was derived. This explanation SHOULD include the parameters required to achieve the stated key strength based on current knowledge of the algorithms.

(Note: Although it is difficult to define what "comparable effort" and "typical block cipher" exactly mean, reasonable approximations are sufficient here. Refer to e.g. [SILVERMAN] for more discussion.)

The key strength depends on the methods used to derive the keys. For instance, if keys are derived from a shared secret (such as a password or a long-term secret), and possibly some public information such as nonces, the effective key strength is limited by the strength of the long-term secret (assuming that the derivation procedure is computationally simple). To take another example, when using public key algorithms, the strength of the symmetric key depends on the strength of the public keys used.

[d] Description of key hierarchy. EAP methods deriving keys MUST either provide a reference to a key hierarchy specification, or describe how Master Session Keys (MSKs) and Extended Master Session Keys (EMSKs) are to be derived.

[e] Indication of vulnerabilities. In addition to the security claims that are made, the specification MUST indicate which of the security claims detailed in Section 7.2.1 are NOT being made.

A.3 hostapd

Inhalt der *hostapd* Konfigurationsdatei `hostspd.conf`:

```
##### hostapd configuration file ##############################################

interface=wlan0
driver=nl80211
```

```
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=0

ctrl_interface=/var/run/hostapd
ctrl_interface_group=0

ssid=bcatestap

country_code=DE
ieee80211d=1
hw_mode=g
channel=1

macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wmm_enabled=1
ieee80211n=1
ht_capab=[HT40][SHORT-GI-20][DSSS_CCK-40]
ieee8021x=1
eapol_version=2
eap_message=ping-from-hostapd
eap_server=1
wpa=3
wpa_key_mgmt=WPA-EAP
wpa_pairwise=CCMP

eap_user_file=/home/pi/hostap/hostapd/hostapd.eap_user

# EAP-BCA
eap_bca_auth_private_key=0291
    ↪ e444ecb0cba525a13b490c25ab5b05d08e00590538e744628fe085e180d1
eap_bca_eth_ipc_file_path=/home/pi/.ethereum/testnet/geth.ipc
eap_bca_eth_auth_address=08aef3ccd9a2f8b73de99ace3cd35aa80f22f88f
eap_bca_eth_auth_passphrase=""
```

Inhalt der *hostapd* EAP-User-Konfigurationsdatei `hostspd.eap_user`:

```
# hostapd user database for integrated EAP server

"testNetworkId"     BCA     55110407995624598
    ↪ c6a935c57e31aa0a04d449425f98e1db744c0630a7a6799a6ec82f1
```

A.4 wpa_supplicant

Inhalt der *wpa_supplicant* Konfigurationsdatei `wpa_supplicant.conf`:

```
##### Example wpa_supplicant configuration file ################################

ctrl_interface=/var/run/wpa_supplicant

# IEEE 802.1X/EAPOL version
# wpa_supplicant is implemented based on IEEE Std 802.1X-2004 which defines
# EAPOL version 2. However, there are many APs that do not handle the new
# version number correctly (they seem to drop the frames completely). In order
# to make wpa_supplicant interoperate with these APs, the version number is set
# to 1 by default. This configuration value can be used to set it to the new
# version (2).
# Note: When using MACsec, eapol_version shall be set to 3, which is
# defined in IEEE Std 802.1X-2010.
```

```
eapol_version=1

ap_scan=1
fast_reauth=1

country=DE
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev

network={
    ssid="bcatestap"
    key_mgmt=WPA-EAP
    eap=BCA
    identity="testNetworkId"
    password=7b4b33b430a80f4f05bab3dde8ff3efb0357adc730c8170ca548dbafb5bbb2b7
        ↪ f3ad03543fb292608aa39dda5f44548e9c4c3390c6b2f840cfb93c221e9f22f7
}
```